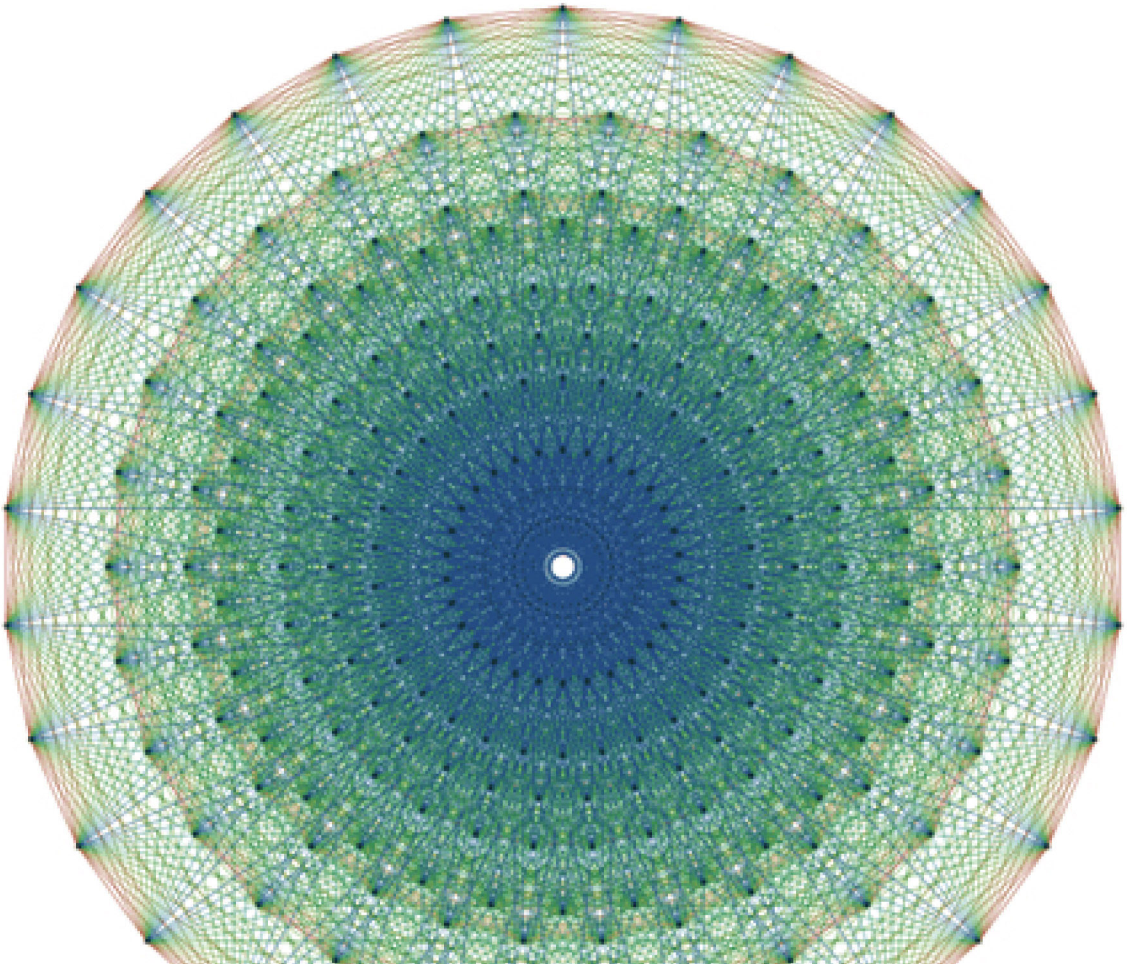
An Introduction to Theory of Computation



Sherwood Malone

First Edition, 2012

ISBN 978-81-323-1707-4

Published by:
Learning Press
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Theory of Computation

The **theory of computation** or **computer theory** is the branch of computer science and mathematics that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into two major branches: computability theory and complexity theory, but both branches deal with formal models of computation.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine. Computer scientists study the Turing machine because it is simple to formulate, can be analyzed and used to prove results, and because it represents what many consider the most powerful possible "reasonable" model of computation. It might seem that the potentially infinite memory capacity is an unrealizable attribute, but any decidable problem solved by a Turing machine will always require only a finite amount of memory. So in principle, any problem that can be solved (decided) by a Turing machine can be solved by a computer that has a bounded amount of memory.

History

The theory of computation can be considered the creation of models of all kinds in the field of computer science. Therefore mathematics and logic are used. In the last century it became an independent academic discipline and was separated from mathematics.

Some pioneers of the theory of computation were Alonzo Church, Alan Turing, Stephen Kleene, John von Neumann, Claude Shannon, and Noam Chomsky.

Computability theory

Computability theory deals primarily with the question of the extent to which a problem is solvable on a computer. The statement that the halting problem cannot be solved by a Turing machine is one of the most important results in computability theory, as it is an example of a concrete problem that is both easy to formulate and impossible to solve using a Turing machine. Much of computability theory builds on the halting problem result.

Another important step in computability theory was Rice's theorem, which states that for all non-trivial properties of partial functions, it is undecidable whether a Turing machine computes a partial function with that property.

Computability theory is closely related to the branch of mathematical logic called recursion theory, which removes the restriction of studying only models of computation which are reducible to the Turing model. Many mathematicians and computational theorists who study recursion theory will refer to it as computability theory.

Complexity theory

Complexity theory considers not only whether a problem can be solved at all on a computer, but also how efficiently the problem can be solved. Two major aspects are considered: time complexity and space complexity, which are respectively how many steps does it take to perform a computation, and how much memory is required to perform that computation.

In order to analyze how much time and space a given algorithm requires, computer scientists express the time or space required to solve the problem as a function of the size of the input problem. For example, finding a particular number in a long list of numbers becomes harder as the list of numbers grows larger. If we say there are n numbers in the list, then if the list is not sorted or indexed in any way we may have to look at every number in order to find the number we're seeking. We thus say that in order to solve this problem, the computer needs to perform a number of steps that grows linearly in the size of the problem.

To simplify this problem, computer scientists have adopted Big O notation, which allows functions to be compared in a way that ensures that particular aspects of a machine's construction do not need to be considered, but rather only the asymptotic behavior as problems become large. So in our previous example we might say that the problem requires $O(n)$ steps to solve.

Perhaps the most important open problem in all of computer science is the question of whether a certain broad class of problems denoted NP can be solved efficiently. This is discussed further at Complexity classes P and NP.

Other formal definitions of computation

Aside from a Turing machine, other equivalent models of computation are in use.

Lambda calculus

> A computation consists of an initial lambda expression (or two if you want to separate the function and its input) plus a finite sequence of lambda terms, each deduced from the preceding term by one application of Beta reduction.

Combinatory logic

is a concept which has many similarities to λ-calculus, but also important differences exist (e.g. fixed point combinator **Y** has normal form in combinatory logic but not in λ-calculus). Combinatory logic was developed with great ambitions: understanding the nature of paradoxes, making foundations of mathematics more economic (conceptually), eliminating the notion of variables (thus clarifying their role in mathematics).

mu-recursive functions

a computation consists of a mu-recursive function, *i.e.* its defining sequence, any input value(s) and a sequence of recursive functions appearing in the defining sequence with inputs and outputs. Thus, if in the defining sequence of a recursive function $f(x)$ the functions $g(x)$ and $h(x,y)$ appear, then terms of the form 'g(5)=7' or 'h(3,2)=10' might appear. Each entry in this sequence needs to be an application of a basic function or follow from the entries above by using composition, primitive recursion or mu recursion. For instance if $f(x) = h(x,g(x))$, then for 'f(5)=3' to appear, terms like 'g(5)=6' and 'h(5,6)=3' must occur above. The computation terminates only if the final term gives the value of the recursive function applied to the inputs.

Markov algorithm

a string rewriting system that uses grammar-like rules to operate on strings of symbols.

Register machine

is a theoretically interesting idealization of a computer. There are several variants. In most of them, each register can hold a natural number (of unlimited size), and the instructions are simple (and few in number), e.g. only decrementation (combined with conditional jump) and incrementation exist (and halting). The lack of the infinite (or dynamically growing) external store (seen at Turing machines) can be understood by replacing its role with Gödel numbering techniques: the fact that each register holds a natural number allows the possibility of representing a complicated thing (e.g. a sequence, or a matrix etc.) by an appropriate huge natural number — unambiguity of both representation and interpretation can be established by number theoretical foundations of these techniques.

P′′

Like Turing machines, P′′ uses an infinite tape of symbols (without random access), and a rather minimalistic set of instructions. But these instructions are very different, thus, unlike Turing machines, P′′ does not need to maintain a distinct state, because all “memory-like” functionality can be provided only by the tape. Instead of rewriting the current symbol, it can perform a modular arithmetic incrementation on it. P′′ has also a pair of instructions for a cycle, inspecting the blank symbol. Despite its minimalistic nature, it has become the parental formal language of an implemented and (for entertainment) used programming language called Brainfuck.

In addition to the general computational models, some simpler computational models are useful for special, restricted applications. Regular expressions, for example, specify string patterns in many contexts, from office productivity software to programming languages.

Another formalism mathematically equivalent to regular expressions, Finite automata are used in circuit design and in some kinds of problem-solving. Context-free grammars specify programming language syntax. Non-deterministic pushdown automata are another formalism equivalent to context-free grammars. Primitive recursive functions are a defined subclass of the recursive functions.

Different models of computation have the ability to do different tasks. One way to measure the power of a computational model is to study the class of formal languages that the model can generate; in such a way to the Chomsky hierarchy of languages is obtained.

Theoretical foundation of computing

A wider view of what constitutes a theory of computation could include topics such as formal semantics and mathematical logic and would include the theoretical aspects of topics such as parallel computation, neural networks and quantum computing.

Chapter 2

Computability Theory

Computability theory, also called **recursion theory**, is a branch of mathematical logic that originated in the 1930s with the study of computable functions and Turing degrees. The field has grown to include the study of generalized computability and definability. In these areas, recursion theory overlaps with proof theory and effective descriptive set theory.

The basic questions addressed by recursion theory are "What does it mean for a function from the natural numbers to themselves to be computable?" and "Can noncomputable functions be classified into a hierarchy based on their level of noncomputability?". The answers to these questions have led to a rich theory that is still being actively researched.

The field is also closely related to computer science. Recursion theorists in mathematical logic often study the theory of relative computability, reducibility notions and degree structures described here. This contrasts with the theory of subrecursive hierarchies, formal methods and formal languages that is common in the study of computability theory in computer science. There is considerable overlap in knowledge and methods between these two research communities, however, and no firm line can be drawn between them.

Computable and uncomputable sets

Recursion theory originated with work of Kurt Gödel, Alonzo Church, Alan Turing, Stephen Kleene and Emil Post in the 1930s.

The fundamental results the researchers obtained established Turing computability as the correct formalization of the informal idea of effective calculation. These results led Stephen Kleene (1952) to coin the two names "Church's thesis" (Kleene 1952:300) and "Turing's Thesis" (p. 376). Nowadays these are often considered as a single hypothesis, the **Church–Turing thesis**, which states that any function that is computable by an algorithm is a computable function. Although initially skeptical, by 1946 Gödel argued in favor of this thesis.

"Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness (or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in giving an absolute notion to an interesting epistemological notion, i.e., one not depending on the formalism chosen."(Gödel 1946 in Davis 1965: 84)

With a definition of effective calculation came the first proofs that there are problems in mathematics that cannot be effectively decided. Church (1936a, 1936b) and Turing (1936), inspired by techniques used by Gödel (1931) to prove his incompleteness theorems, independently demonstrated that the Entscheidungsproblem is not effectively decidable. This result showed that there is no algorithmic procedure that can correctly decide whether arbitrary mathematical propositions are true or false.

Many problems of mathematics have been shown to be undecidable after these initial examples were established. In 1947, Markov and Post published independent papers showing that the word problem for semigroups cannot be effectively decided. Extending this result, Pyotr Novikov and William Boone showed independently in the 1950s that the word problem for groups is not effectively solvable: there is no effective procedure that, given a word in a finitely presented group, will decide whether the element represented by the word is the identity element of the group. In 1970, Yuri Matiyasevich proved Matiyasevich's theorem, which implies that Hilbert's tenth problem has no effective solution; this problem asked whether there is an effective procedure to decide whether a Diophantine equation over the integers has a solution in the integers. The list of undecidable problems gives additional examples of problems with no computable solution.

The study of which mathematical constructions can be effectively performed is sometimes called **recursive mathematics**; the *Handbook of Recursive Mathematics* (Ershov *et al.* 1998) covers many of the known results in this field.

Turing computability

The main form of computability studied in recursion theory was introduced by Turing (1936). A set of natural numbers is said to be a **computable set** (also called a **decidable**, **recursive**, or **Turing computable** set) if there is a Turing machine that, given a number *n*, halts with output 1 if *n* is in the set and halts with output 0 if *n* is not in the set. A function *f* from the natural numbers to themselves is a **recursive** or **(Turing) computable function** if there is a Turing machine that, on input *n*, halts and returns output *f*(*n*). The use of Turing machines here is not necessary; there are many other models of computation that have the same computing power as Turing machines; for example the μ-recursive functions obtained from primitive recursion and the μ operator.

The terminology for recursive functions and sets is not completely standardized. The definition in terms of μ-recursive functions as well as a different definition of *rekursiv* functions by Gödel led to the traditional name *recursive* for sets and functions computable by a Turing machine. The word **decidable** stems from the German word **Entscheidungsproblem** which was used in the original papers of Turing and others. In contemporary use, the term "computable function" has various definitions: according to Cutland (1980), it is a partial recursive function (which can be undefined for some inputs), while according to Soare (1987) it is a total recursive (equivalently, general recursive) function. This article follows the second of these conventions. Soare (1996) gives additional comments about the terminology.

Not every set of natural numbers is computable. The halting problem, which is the set of (descriptions of) Turing machines that halt on input 0, is a well known example of a noncomputable set. The existence of many noncomputable sets follows from the facts that there are only countably many Turing machines, and thus only countably many computable sets, but there are uncountably many sets of natural numbers.

Although the Halting problem is not computable, it is possible to simulate program execution and produce an infinite list of the programs that do halt. Thus the halting problem is an example of a **recursively enumerable set**, which is a set that can be enumerated by a Turing machine (other terms for recursively enumerable include **computably enumerable** and **semidecidable**). Equivalently, a set is recursively enumerable if and only if it is the range of some computable function. The recursively enumerable sets, although not decidable in general, have been studied in detail in recursion theory.

Areas of research in recursion theory

Beginning with the theory of recursive sets and functions described above, the field of recursion theory has grown to include the study of many closely related topics. These are not independent areas of research: each of these areas draws ideas and results from the others, and most recursion theorists are familiar with the majority of them.

Relative computability and the Turing degrees

Recursion theory in mathematical logic has traditionally focused on **relative computability**, a generalization of Turing computability defined using oracle Turing machines, introduced by Turing (1939). An oracle Turing machine is a hypothetical device which, in addition to performing the actions of a regular Turing machine, is able to ask questions of an **oracle**, which is a particular set of natural numbers. The oracle machine may only ask questions of the form "Is *n* in the oracle set?". Each question will be immediately answered correctly, even if the oracle set is not computable. Thus an oracle machine with a noncomputable oracle will be able to compute sets that are not computable without an oracle.

Informally, a set of natural numbers *A* is **Turing reducible** to a set *B* if there is an oracle machine that correctly tells whether numbers are in *A* when run with *B* as the oracle set (in this case, the set *A* is also said to be (**relatively**) **computable from** *B* and **recursive in** *B*). If a set *A* is Turing reducible to a set *B* and *B* is Turing reducible to *A* then the sets are said to have the same **Turing degree** (also called **degree of unsolvability**). The Turing degree of a set gives a precise measure of how uncomputable the set is.

The natural examples of sets that are not computable, including many different sets that encode variants of the halting problem, have two properties in common:

1. They are recursively enumerable, and
2. Each can be translated into any other via a many-one reduction. That is, given such sets A and B, there is a total computable function f such that $A = \{x : f(x) \in B\}$. These sets are said to be **many-one equivalent** (or **m-equivalent**).

Many-one reductions are "stronger" than Turing reductions: if a set A is many-one reducible to a set B, then A is Turing reducible to B, but the converse does not always hold. Although the natural examples of noncomputable sets are all many-one equivalent, it is possible to construct recursively enumerable sets A and B such that A is Turing reducible to B but not many-one reducible to B. It can be shown that every recursively enumerable set is many-one reducible to the halting problem, and thus the halting problem is the most complicated recursively enumerable set with respect to many-one reducibility and with respect to Turing reducibility. Post (1944) asked whether *every* recursively enumerable set is either computable or Turing equivalent to the halting problem, that is, whether there is no recursively enumerable set with a Turing degree intermediate between those two.

As intermediate results, Post defined natural types of recursively enumerable sets like the simple, hypersimple and hyperhypersimple sets. Post showed that these sets are strictly between the computable sets and the halting problem with respect to many-one reducibility. Post also showed that some of them are strictly intermediate under other reducibility notions stronger than Turing reducibility. But Post left open the main problem of the existence of recursively enumerable sets of intermediate Turing degree; this problem became known as **Post's problem**. After ten years, Kleene and Post showed in 1954 that there are intermediate Turing degrees between those of the computable sets and the halting problem, but they failed to show that any of these degrees contains a recursively enumerable set. Very soon after this, Friedberg and Muchnik independently solved Post's problem by establishing the existence of recursively enumerable sets of intermediate degree. This groundbreaking result opened a wide study of the Turing degrees of the recursively enumerable sets which turned out to possess a very complicated and non-trivial structure.

There are uncountably many sets that are not recursively enumerable, and the investigation of the Turing degrees of all sets is as central in recursion theory as the investigation of the recursively enumerable Turing degrees. Many degrees with special properties were constructed: **hyperimmune-free degrees** where every function computable relative to that degree is majorized by a (unrelativized) computable function; **high degrees** relative to which one can compute a function f which dominates every computable function g in the sense that there is a constant c depending on g such that $g(x) < f(x)$ for all $x > c$; **random degrees** containing algorithmically random sets; **1-generic** degrees of 1-generic sets; and the degrees below the halting problem of limit-recursive sets.

The study of arbitrary (not necessarily recursively enumerable) Turing degrees involves the study of the Turing jump. Given a set *A*, the **Turing jump** of *A* is a set of natural numbers encoding a solution to the halting problem for oracle Turing machines running with oracle *A*. The Turing jump of any set is always of higher Turing degree than the original set, and a theorem of Friedburg shows that any set that computes the Halting problem can be obtained as the Turing jump of another set. Post's theorem establishes a close relationship between the Turing jump operation and the arithmetical hierarchy, which is a classification of certain subsets of the natural numbers based on their definability in arithmetic.

Much recent research on Turing degrees has focused on the overall structure of the set of Turing degrees and the set of Turing degrees containing recursively enumerable sets. A deep theorem of Shore and Slaman (1999) states that the function mapping a degree *x* to the degree of its Turing jump is definable in the partial order of the Turing degrees. A recent survey by Ambos-Spies and Fejer (2006) gives an overview of this research and its historical progression.

Other reducibilities

An ongoing area of research in recursion theory studies reducibility relations other than Turing reducibility. Post (1944) introduced several **strong reducibilities**, so named because they imply truth-table reducibility. A Turing machine implementing a strong reducibility will compute a total function regardless of which oracle it is presented with. **Weak reducibilities** are those where a reduction process may not terminate for all oracles; Turing reducibility is one example.

The strong reducibilities include:

- One-one reducibility: *A* is **one-one reducible** (or **1-reducible**) to *B* if there is a total computable injective function *f* such that each *n* is in *A* if and only if *f*(*n*) is in *B*.
- Many-one reducibility: This is essentially one-one reducibility without the constraint that *f* be injective. *A* is **many-one reducible** (or **m-reducible**) to *B* if there is a total computable function *f* such that each *n* is in *A* if and only if *f*(*n*) is in *B*.
- Truth-table reducibility: *A* is truth-table reducible to *B* if *A* is Turing reducible to *B* via an oracle Turing machine that computes a total function regardless of the oracle it is given. Because of compactness of Cantor space, this is equivalent to saying that the reduction presents a single list of questions (depending only on the input) to the oracle simultaneously, and then having seen their answers is able to produce an output without asking additional questions regardless of the oracle's answer to the initial queries. Many variants of truth-table reducibility have also been studied.

Further reducibilities (positive, disjunctive, conjunctive, linear and their weak and bounded versions) are discussed in Reduction (recursion theory).

The major research on strong reducibilities has been to compare their theories, both for the class of all recursively enumerable sets as well as for the class of all subsets of the natural numbers. Furthermore, the relations between the reducibilities has been studied. For example, it is known that every Turing degree is either a truth-table degree or is the union of infinitely many truth-table degrees.

Reducibilities weaker than Turing reducibility (that is, reducibilities that are implied by Turing reducibility) have also been studied. The most well known are arithmetical reducibility and hyperarithmetical reducibility. These reducibilities are closely connected to definability over the standard model of arithmetic.

Rice's theorem and the arithmetical hierarchy

Rice showed that for every nontrivial class C (which contains some but not all r.e. sets) the index set $E = \{e$: the eth r.e. set W_e is in $C\}$ has the property that either the halting problem or its complement is many-one reducible to E, that is, can be mapped using a many-one reduction to E. But, many of these index sets are even more complicated than the halting problem. These type of sets can be classified using the arithmetical hierarchy. For example, the index set FIN of class of all finite sets is on the level Σ_2, the index set REC of the class of all recursive sets is on the level Σ_3, the index set COFIN of all cofinite sets is also on the level Σ_3 and the index set COMP of the class of all Turing-complete sets Σ_4. These hierarchy levels are defined inductively, Σ_{n+1} contains just all sets which are recursively enumerable relative to Σ_n; Σ_1 contains the recursively enumerable sets. The index sets given here are even complete for their levels, that is, all the sets in these levels can be many-one reduced to the given index sets.

Reverse mathematics

The program of **reverse mathematics** asks which set-existence axioms are necessary to prove particular theorems of mathematics in subsystems of second-order arithmetic. This study was initiated by Harvey Friedman and was studied in detail by Stephen Simpson and others; Simpson (1999) gives a detailed discussion of the program. The set-existence axioms in question correspond informally to axioms saying that the powerset of the natural numbers is closed under various reducibility notions. The weakest such axiom studied in reverse mathematics is **recursive comprehension**, which states that the powerset of the naturals is closed under Turing reducibility.

Numberings

A numbering is an enumeration of functions; it has two parameters, e and x and outputs the value of the e-th function in the numbering on the input x. Numberings can be partial-recursive although some of its members are total recursive, that is, computable functions. Acceptable or Gödel numberings are those into which all others can be translated. A Friedberg numbering (named after its discoverer) is a one-one numbering of all partial-recursive functions; it is necessarily not an acceptable numbering. Later research dealt also with numberings of other classes like classes of recursively enumerable sets.

Goncharov discovered for example a class of recursively enumerable sets for which the numberings fall into exactly two classes with respect to recursive isomorphisms.

The priority method

Post's problem was solved with a method called the **priority method**; a proof using this method is called a **priority argument**. This method is primarily used to construct recursively enumerable sets with particular properties. To use this method, the desired properties of the set to be constructed are broken up into an infinite list of goals, known as **requirements**, so that satisfying all the requirements will cause the set constructed to have the desired properties. Each requirement is assigned to a natural number representing the priority of the requirement; so 0 is assigned to the most important priority, 1 to the second most important, and so on. The set is then constructed in stages, each stage attempting to satisfy one of more of the requirements by either adding numbers to the set or banning numbers from the set so that the final set will satisfy the requirement. It may happen that satisfying one requirement will cause another to become unsatisfied; the priority order is used to decide what to do in such an event.

Priority arguments have been employed to solve many problems in recursion theory, and have been classified into a hierarchy based on their complexity (Soare 1987). Because complex priority arguments can be technical and difficult to follow, it has traditionally been considered desirable to prove results without priority arguments, or to see if results proved with priority arguments can also be proved without them. For example, Kummer published a paper on a proof for the existence of Friedberg numberings without using the priority method.

The lattice of recursively enumerable sets

When Post defined the notion of a simple set as an r.e. set with an infinite complement not containing any infinite r.e. set, he started to study the structure of the recursively enumerable sets under inclusion. This lattice became a well-studied structure. Recursive sets can be defined in this structure by the basic result that a set is recursive if and only if the set and its complement are both recursively enumerable. Infinite r.e. sets have always infinite recursive subsets; but on the other hand, simple sets exist but do not have a coinfinite recursive superset. Post (1944) introduced already hypersimple and hyperhypersimple sets; later maximal sets were constructed which are r.e. sets such that every r.e. superset is either a finite variant of the given maximal set or is co-finite. Post's original motivation in the study of this lattice was to find a structural notion such that every set which satisfies this property is neither in the Turing degree of the recursive sets nor in the Turing degree of the halting problem. Post did not find such a property and the solution to his problem applied priority methods instead; Harrington and Soare (1991) found eventually such a property.

Automorphism problems

Another important question is the existence of automorphisms in recursion-theoretic structures. One of these structures is that one of recursively enumerable sets under inclusion modulo finite difference; in this structure, A is below B if and only if the set difference $B - A$ is finite. Maximal sets (as defined in the previous paragraph) have the property that they cannot be automorphic to non-maximal sets, that is, if there is an automorphism of the recursive enumerable sets under the structure just mentioned, then every maximal set is mapped to another maximal set. Soare (1974) showed that also the converse holds, that is, every two maximal sets are automorphic. So the maximal sets form an orbit, that is, every automorphism preserves maximality and any two maximal sets are transformed into each other by some automorphism. Harrington gave a further example of an automorphic property: that of the creative sets, the sets which are many-one equivalent to the halting problem.

Besides the lattice of recursively enumerable sets, automorphisms are also studied for the structure of the Turing degrees of all sets as well as for the structure of the Turing degrees of r.e. sets. In both cases, Cooper claims to have constructed nontrivial automorphisms which map some degrees to other degrees; this construction has, however, not been verified and some colleagues believe that the construction contains errors and that the question of whether there is a nontrivial automorphism of the Turing degrees is still one of the main unsolved questions in this area (Slaman and Woodin 1986, Ambos-Spies and Fejer 2006).

Kolmogorov complexity

The field of Kolmogorov complexity and algorithmic randomness was developed during the 1960s and 1970s by Chaitin, Kolmogorov, Levin, Martin-Löf and Solomonoff (the names are given here in alphabetical order; much of the research was independent, and the unity of the concept of randomness was not understood at the time). The main idea is to consider a universal Turing machine U and to measure the complexity of a number (or string) x as the length of the shortest input p such that $U(p)$ outputs x. This approach revolutionized earlier ways to determine when an infinite sequence (equivalently, characteristic function of a subset of the natural numbers) is random or not by invoking a notion of randomness for finite objects. Kolmogorov complexity became not only a subject of independent study but is also applied to other subjects as a tool for obtaining proofs. There are still many open problems in this area. For that reason, a recent research conference in this area was held in January 2007 and a list of open problems is maintained by Joseph Miller and Andre Nies.

Frequency computation

This branch of recursion theory analyzed the following question: For fixed m and n with $0 < m < n$, for which functions A is it possible to compute for any different n inputs $x_1, x_2, ..., x_n$ a tuple of n numbers $y_1,y_2,...,y_n$ such that at least m of the equations $A(x_k) = y_k$ are true. Such sets are known as (m, n)-recursive sets. The first major result in this branch of Recursion Theory is Trakhtenbrot's result that a set is computable if it is (m, n)-recursive for some m, n with $2m > n$. On the other hand, Jockusch's semirecursive sets (which were already known informally before Jockusch introduced them 1968) are examples of a set which is (m, n)-recursive if and only if $2m < n + 1$. There are uncountably many of these sets and also some recursively enumerable but noncomputable sets of this type. Later, Degtev established a hierarchy of recursively enumerable sets that are $(1, n + 1)$-recursive but not $(1, n)$-recursive. After a long phase of research by Russian scientists, this subject became repopularized in the west by Beigel's thesis on bounded queries, which linked frequency computation to the above mentioned bounded reducibilities and other related notions. One of the major results was Kummer's Cardinality Theory which states that a set A is computable if and only if there is an n such that some algorithm enumerates for each tuple of n different numbers up to n many possible choices of the cardinality of this set of n numbers intersected with A; these choices must contain the true cardinality but leave out at least one false one.

Inductive inference

This is the recursion-theoretic branch of learning theory. It is based on Gold's model of learning in the limit from 1967 and has developed since then more and more models of learning. The general scenario is the following: Given a class S of computable functions, is there a learner (that is, recursive functional) which outputs for any input of the form $(f(0),f(1),...,f(n))$ a hypothesis. A learner M learns a function f if almost all hypotheses are the same index e of f with respect to a previously agreed on acceptable numbering of all computable functions; M learns S if M learns every f in S. Basic results are that all recursively enumerable classes of functions are learnable while the class REC of all computable functions is not learnable. Many related models have been considered and also the learning of classes of recursively enumerable sets from positive data is a topic studied from Gold's pioneering paper in 1967 onwards.

Generalizations of Turing computability

Recursion theory includes the study of generalized notions of this field such as arithmetic reducibility, hyperarithmetical reducibility and α-recursion theory, as described by Sacks (1990). These generalized notions include reducibilities that cannot be executed by Turing machines but are nevertheless natural generalizations of Turing reducibility. These studies include approaches to investigate the analytical hierarchy which differs from the arithmetical hierarchy by permitting quantification over sets of natural numbers in addition to quantification over individual numbers. These areas are linked to the theories of well-orderings and trees; for example the set of all indices of recursive (nonbinary) trees without infinite branches is complete for level $\Pi^1_1$of the analytical hierarchy. Both Turing reducibility and hyperarithmetical reducibility are important in the field of effective descriptive set theory. The even more general notion of degrees of constructibility is studied in set theory.

Continuous computability theory

Computability theory for digital computation is well developed. Computability theory is less well developed for analog computation that occurs in analog computers, analog signal processing, analog electronics, neural networks and continuous-time control theory, modelled by differential equations and continuous dynamical systems.

Relationships between definability, proof and computability

There are close relationships between the Turing degree of a set of natural numbers and the difficulty (in terms of the arithmetical hierarchy) of defining that set using a first-order formula. One such relationship is made precise by Post's theorem. A weaker relationship was demonstrated by Kurt Gödel in the proofs of his completeness theorem and incompleteness theorems. Gödel's proofs show that the set of logical consequences of an effective first-order theory is a recursively enumerable set, and that if the theory is strong enough this set will be uncomputable. Similarly, Tarski's indefinability theorem can be interpreted both in terms of definability and in terms of computability.

Recursion theory is also linked to second order arithmetic, a formal theory of natural numbers and sets of natural numbers. The fact that certain sets are computable or relatively computable often implies that these sets can be defined in weak subsystems of second order arithmetic. The program of reverse mathematics uses these subsystems to measure the noncomputability inherent in well known mathematical theorems. Simpson (1999) discusses many aspects of second-order arithmetic and reverse mathematics.

The field of proof theory includes the study of second-order arithmetic and Peano arithmetic, as well as formal theories of the natural numbers weaker than Peano arithmetic. One method of classifying the strength of these weak systems is by characterizing which computable functions the system can prove to be total. For example, in primitive recursive arithmetic any computable function that is provably total is actually primitive recursive, while Peano arithmetic proves that functions like the Ackerman function, which are not primitive recursive, are total. Not every total computable function is provably total in Peano arithmetic, however; an example of such a function is provided by Goodstein's theorem.

Chapter 3

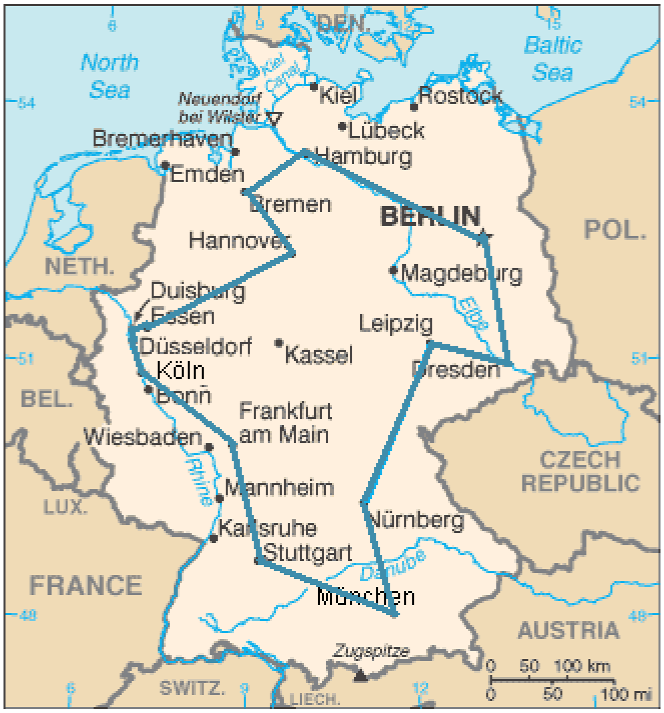
Computational Complexity Theory

Computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty. In this context, a computational problem is understood to be a task that is in principle amenable to being solved by a computer. Informally, a computational problem consists of problem instances and solutions to these problem instances. For example, primality testing is the problem of determining whether a given number is prime or not. The instances of this problem are natural numbers, and the solution to an instance is *yes* or *no* based on whether the number is prime or not.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). One of the roles of computational complexity theory is to determine the practical limits on what computers can and cannot do.

Closely related fields in theoretical computer science are analysis of algorithms and computability theory. A key distinction between computational complexity theory and analysis of algorithms is that the latter is devoted to analyzing the amount of resources needed by a particular algorithm to solve a problem, whereas the former asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, it tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can be solved in principle algorithmically.

Computational problems



An optimal traveling salesperson tour through Germany’s 15 largest cities. It is the shortest among 43,589,145,600 possible tours visiting each city exactly once.

Problem instances

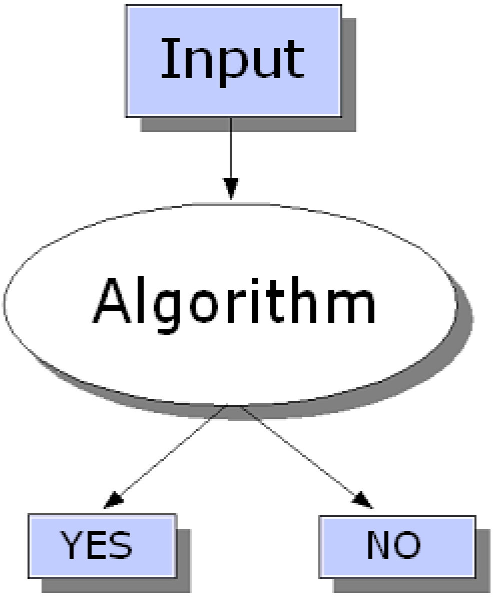
A computational problem can be viewed as an infinite collection of *instances* together with a *solution* for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself. In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem. For example, consider the problem of primality testing. The instance is a number and the solution is "yes" if the number is prime and "no" otherwise. Alternately, the instance is a particular input to the problem, and the solution is the output corresponding to the given input.

To further highlight the difference between a problem and an instance, consider the following instance of the decision version of the traveling salesman problem: Is there a route of length at most 2000 kilometres passing through all of Germany's 15 largest cities? The answer to this particular problem instance is of little use for solving other instances of the problem, such as asking for a round trip through all sights in Milan whose total length is at most 10 km. For this reason, complexity theory addresses computational problems and not particular problem instances.

Representing problem instances

When considering computational problems, a problem instance is a string over an alphabet. Usually, the alphabet is taken to be the binary alphabet (i.e., the set {0,1}), and thus the strings are bitstrings. As in a real-world computer, mathematical objects other than bitstrings must be suitably encoded. For example, integers can be represented in binary notation, and graphs can be encoded directly via their adjacency matrices, or by encoding their adjacency lists in binary.

Even though some proofs of complexity-theoretic theorems regularly assume some concrete choice of input encoding, one tries to keep the discussion abstract enough to be independent of the choice of encoding. This can be achieved by ensuring that different representations can be transformed into each other efficiently.



A decision problem has only two possible outputs, *yes* or *no* (or alternately 1 or 0) on any input.

Decision problems as formal languages

Decision problems are one of the central objects of study in computational complexity theory. A decision problem is a special type of computational problem whose answer is either *yes* or *no*, or alternately either 1 or 0. A decision problem can be viewed as a formal language, where the members of the language are instances whose answer is yes, and the non-members are those instances whose output is no. The objective is to decide, with the aid of an algorithm, whether a given input string is member of the formal language under consideration. If the algorithm deciding this problem returns the answer *yes*, the algorithm is said to accept the input string, otherwise it is said to reject the input.

An example of a decision problem is the following. The input is an arbitrary graph. The problem consists in deciding whether the given graph is connected, or not. The formal language associated with this decision problem is then the set of all connected graphs—of course, to obtain a precise definition of this language, one has to decide how graphs are encoded as binary strings.

Function problems

A function problem is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just yes or no. Notable examples include the traveling salesman problem and the integer factorization problem.

It is tempting to think that the notion of function problems is much richer than the notion of decision problems. However, this is not really the case, since function problems can be recast as decision problems. For example, the multiplication of two integers can be expressed as the set of triples (a, b, c) such that the relation $a \times b = c$ holds. Deciding whether a given triple is member of this set corresponds to solving the problem of multiplying two numbers.

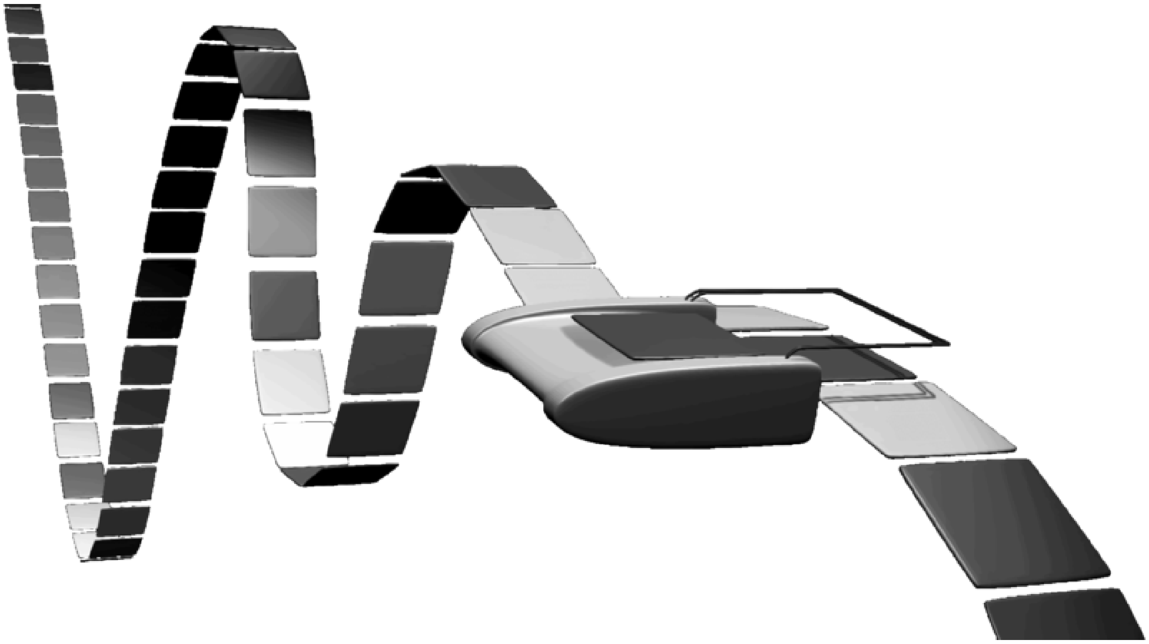
Measuring the size of an instance

To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem. However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve. Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as function of the size of the instance. This is usually taken to be the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the input size. For instance, in the problem of finding whether a graph is connected, how much more time does it take to solve a problem for a graph with $2n$ vertices compared to the time taken for a graph with n vertices?

If the input size is n, the time taken can be expressed as a function of n. Since the time taken on different inputs of the same size can be different, the worst-case time complexity $T(n)$ is defined to be the maximum time taken over all inputs of size n. If $T(n)$ is a polynomial in n, then the algorithm is said to be a polynomial time algorithm. Cobham's thesis says that a problem can be solved with a feasible amount of resources if it admits a polynomial time algorithm.

Machine models and complexity measures

Turing Machine



An artistic representation of a Turing machine

A Turing machine is a mathematical model of a general computing machine. It is a theoretical device that manipulates symbols contained on a strip of tape. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. It is believed that if a problem can be solved by an algorithm, there exists a Turing machine that solves the problem. Indeed, this is the statement of the Church–Turing thesis. Furthermore, it is known that everything that can be computed on other models of computation known to us today, such as a RAM machine, Conway's Game of Life, cellular automata or any programming language can be computed on a Turing machine. Since Turing machines are easy to analyze mathematically, and are believed to be as powerful as any other model of computation, the Turing machine is the most commonly used model in complexity theory.

Many types of Turing machines are used to define complexity classes, such as deterministic Turing machines, probabilistic Turing machines, non-deterministic Turing machines, quantum Turing machines, symmetric Turing machines and alternating Turing machines. They are all equally powerful in principle, but when resources (such as time or space) are bounded, some of these may be more powerful than others.

A deterministic Turing machine is the most basic Turing machine, which uses a fixed set of rules to determine its future actions. A probabilistic Turing machine is a deterministic Turing machine with an extra supply of random bits. The ability to make probabilistic decisions often helps algorithms solve problems more efficiently. Algorithms that use random bits are called randomized algorithms. A non-deterministic Turing machine is a deterministic Turing machine with an added feature of non-determinism, which allows a Turing machine to have multiple possible future actions from a given state. One way to view non-determinism is that the Turing machine branches into many possible computational paths at each step, and if it solves the problem in any of these branches, it is said to have solved the problem. Clearly, this model is not meant to be a physically realizable model, it is just a theoretically interesting abstract machine that gives rise to particularly interesting complexity classes.

Other machine models

Many machine models different from the standard multi-tape Turing machines have been proposed in the literature, for example random access machines. Perhaps surprisingly, each of these models can be converted to another without substantial overhead in time and memory consumption. What all these models have in common is that the machines operate deterministically.

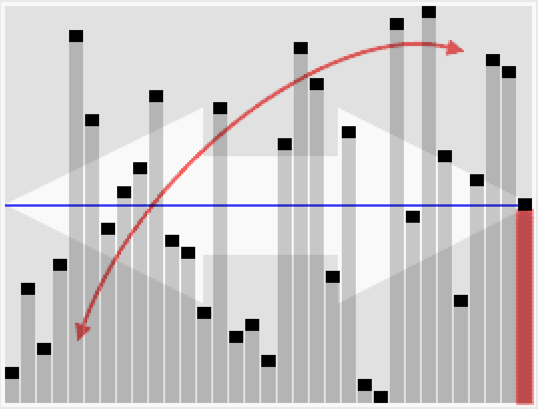
However, some computational problems are easier to analyze in terms of more unusual resources. For example, a nondeterministic Turing machine is a computational model that is allowed to branch out to check many different possibilities at once. The nondeterministic Turing machine has very little to do with how we physically want to compute algorithms, but its branching exactly captures many of the mathematical models we want to analyze, so that nondeterministic time is a very important resource in analyzing computational problems.

Complexity measures

For a precise definition of what it means to solve a problem using a given amount of time and space, a computational model such as the deterministic Turing machine is used. The *time required* by a deterministic Turing machine M on input x is the total number of state transitions, or steps, the machine makes before it halts and outputs the answer ("yes" or "no"). A Turing machine M is said to operate within time $f(n)$, if the time required by M on each input of length n is at most $f(n)$. A decision problem A can be solved in time $f(n)$ if there exists a Turing machine operating in time $f(n)$ that solves the problem. Since complexity theory is interested in classifying problems based on their difficulty, one defines sets of problems based on some criteria. For instance, the set of problems solvable within time $f(n)$ on a deterministic Turing machine is then denoted by DTIME($f(n)$).

Analogous definitions can be made for space requirements. Although time and space are the most well-known complexity resources, any complexity measure can be viewed as a computational resource. Complexity measures are very generally defined by the Blum complexity axioms. Other complexity measures used in complexity theory include communication complexity, circuit complexity, and decision tree complexity.

Best, worst and average case complexity



Visualization of the quicksort algorithm that has average case performance $\Theta(n \log n)$.

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size n.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n.

For example, consider the deterministic sorting algorithm quicksort. This solves the problem of sorting a list of integers that is given as the input. The best-case scenario is when the input is already sorted, and the algorithm takes time $O(n \log n)$ for such inputs. The worst-case is when the input is sorted in reverse order, and the algorithm takes time $O(n^2)$ for this case. If we assume that all possible permutations of the input list are equally likely, the average time taken for sorting is $O(n \log n)$.

Upper and lower bounds on the complexity of problems

To classify the computation time (or similar resources, such as space consumption), one is interested in proving upper and lower bounds on the minimum amount of time required by the most efficient algorithm solving a given problem. The complexity of an algorithm is usually taken to be its worst-case complexity, unless specified otherwise. Analyzing a particular algorithm falls under the field of analysis of algorithms. To show an upper bound $T(n)$ on the time complexity of a problem, one needs to show only that there is a particular algorithm with running time at most $T(n)$. However, proving lower bounds is much more difficult, since lower bounds make a statement about all possible algorithms that solve a given problem. The phrase "all possible algorithms" includes not just the algorithms known today, but any algorithm that might be discovered in the future. To show a lower bound of $T(n)$ for a problem requires showing that no algorithm can have time complexity lower than $T(n)$.

Upper and lower bounds are usually stated using the big Oh notation, which hides constant factors and smaller terms. This makes the bounds independent of the specific details of the computational model used. For instance, if $T(n) = 7n^2 + 15n + 40$, in big Oh notation one would write $T(n) = O(n^2)$.

Complexity classes

Defining complexity classes

A **complexity class** is a set of problems of related complexity. Simpler complexity classes are defined by the following factors:

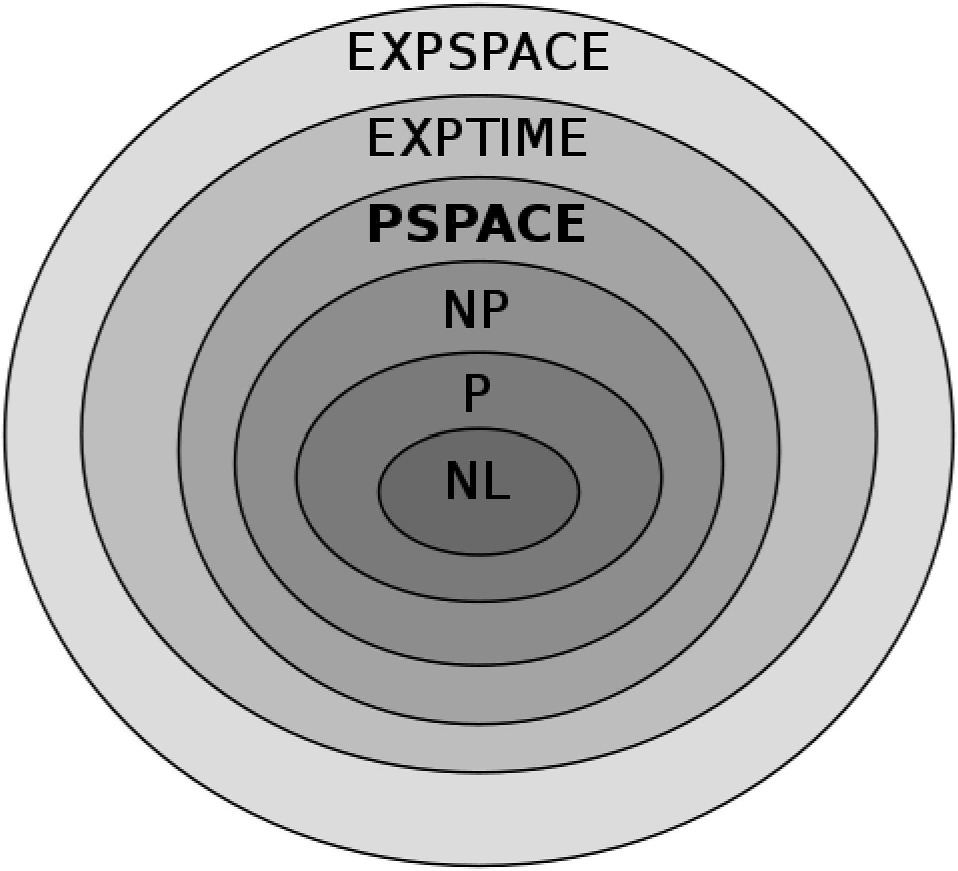
- The type of computational problem: The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, counting problems, optimization problems, promise problems, etc.
- The model of computation: The most common model of computation is the deterministic Turing machine, but many complexity classes are based on nondeterministic Turing machines, Boolean circuits, quantum Turing machines, monotone circuits, etc.
- The resource (or resources) that are being bounded and the bounds: These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

Of course, some complexity classes have complex definitions that do not fit into this framework. Thus, a typical complexity class has a definition like the following:

> The set of decision problems solvable by a deterministic Turing machine within time $f(n)$. (This complexity class is known as DTIME($f(n)$).)

But bounding the computation time above by some concrete function $f(n)$ often yields complexity classes that depend on the chosen machine model. For instance, the language $\{xx \mid x \text{ is any binary string}\}$ can be solved in linear time on a multi-tape Turing machine, but necessarily requires quadratic time in the model of single-tape Turing machines. If we allow polynomial variations in running time, Cobham-Edmonds thesis states that "the time complexities in any two reasonable and general models of computation are polynomially related" (Goldreich 2008, Chapter 1.2). This forms the basis for the complexity class P, which is the set of decision problems solvable by a deterministic Turing machine within polynomial time. The corresponding set of function problems is FP.

Important complexity classes



A representation of the relation among complexity classes

Many important complexity classes can be defined by bounding the time or space used by the algorithm. Some important complexity classes of decision problems defined in this manner are the following:

Complexity class	Model of computation	Resource constraint
DTIME(*f*(*n*))	Deterministic Turing machine	Time *f*(*n*)
P	Deterministic Turing machine	Time poly(*n*)
EXPTIME	Deterministic Turing machine	Time $2^{\text{poly}(n)}$
NTIME(*f*(*n*))	Non-deterministic Turing machine	Time *f*(*n*)
NP	Non-deterministic Turing machine	Time poly(*n*)
NEXPTIME	Non-deterministic Turing machine	Time $2^{\text{poly}(n)}$
DSPACE(*f*(*n*))	Deterministic Turing machine	Space *f*(*n*)
L	Deterministic Turing machine	Space O(log *n*)
PSPACE	Deterministic Turing machine	Space poly(*n*)
EXPSPACE	Deterministic Turing machine	Space $2^{\text{poly}(n)}$

NSPACE($f(n)$)	Non-deterministic Turing machine Space $f(n)$
NL	Non-deterministic Turing machine Space O(log n)
NPSPACE	Non-deterministic Turing machine Space poly(n)
NEXPSPACE	Non-deterministic Turing machine Space $2^{\text{poly}(n)}$

It turns out that PSPACE = NPSPACE and EXPSPACE = NEXPSPACE by Savitch's theorem.

Other important complexity classes include BPP, ZPP and RP, which are defined using probabilistic Turing machines; AC and NC, which are defined using Boolean circuits and BQP and QMA, which are defined using quantum Turing machines. #P is an important complexity class of counting problems (not decision problems). Classes like IP and AM are defined using Interactive proof systems. ALL is the class of all decision problems.

Hierarchy theorems

For the complexity classes defined in this way, it is desirable to prove that relaxing the requirements on (say) computation time indeed defines a bigger set of problems. In particular, although DTIME(n) is contained in DTIME(n^2), it would be interesting to know if the inclusion is strict. For time and space requirements, the answer to such questions is given by the time and space hierarchy theorems respectively. They are called hierarchy theorems because they induce a proper hierarchy on the classes defined by constraining the respective resources. Thus there are pairs of complexity classes such that one is properly included in the other. Having deduced such proper set inclusions, we can proceed to make quantitative statements about how much more additional time or space is needed in order to increase the number of problems that can be solved.

More precisely, the time hierarchy theorem states that

$$\mathrm{DTIME}\big(f(n)\big) \subsetneq \mathrm{DTIME}\big(f(n) \cdot \log^2(f(n))\big).$$

The space hierarchy theorem states that

$$\mathrm{DSPACE}\big(f(n)\big) \subsetneq \mathrm{DSPACE}\big(f(n) \cdot \log(f(n))\big).$$

The time and space hierarchy theorems form the basis for most separation results of complexity classes. For instance, the time hierarchy theorem tells us that P is strictly contained in EXPTIME, and the space hierarchy theorem tells us that L is strictly contained in PSPACE.

Reduction

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem *X* can be solved using an algorithm for *Y*, *X* is no more difficult than *Y*, and we say that *X reduces* to *Y*. There are many different types of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as polynomial-time reductions or log-space reductions.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem *X* is *hard* for a class of problems *C* if every problem in *C* can be reduced to *X*. Thus no problem in *C* is harder than *X*, since an algorithm for *X* allows us to solve any problem in *C*. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems.

If a problem *X* is in *C* and hard for *C*, then *X* is said to be *complete* for *C*. This means that *X* is the hardest problem in *C*. (Since many problems could be equally hard, one might say that *X* is one of the hardest problems in *C*.) Thus the class of NP-complete problems contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. Because the problem P = NP is not solved, being able to reduce a known NP-complete problem, Π_2, to another problem, Π_1, would indicate that there is no known polynomial-time solution for Π_1. This is due to the fact that a polynomial-time solution to Π_1 would yield a polynomial-time solution to Π_2. Similarly, because all NP problems can be reduced to the set, finding an NP-complete problem that can be solved in polynomial time would mean that P = NP.

Important open problems

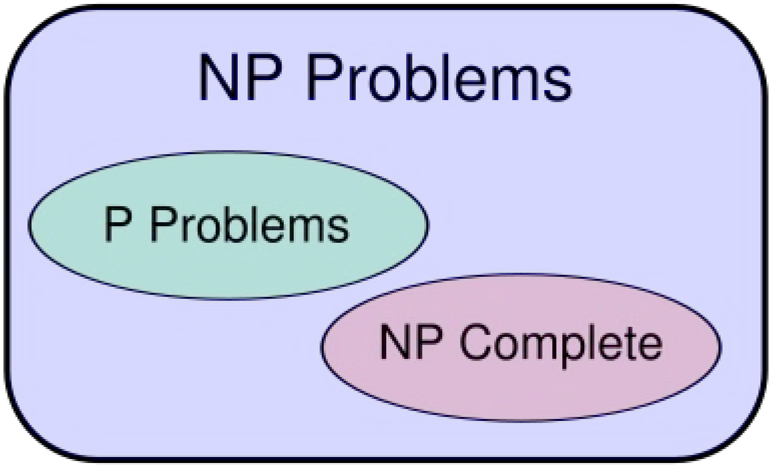


Diagram of complexity classes provided that P ≠ NP. The existence of problems in NP outside both P and NP-complete in this case was established by Ladner.

P versus NP problem

The complexity class P is often seen as a mathematical abstraction modeling those computational tasks that admit an efficient algorithm. This hypothesis is called the Cobham–Edmonds thesis. The complexity class NP, on the other hand, contains many problems that people would like to solve efficiently, but for which no efficient algorithm is known, such as the Boolean satisfiability problem, the Hamiltonian path problem and the vertex cover problem. Since deterministic Turing machines are special nondeterministic Turing machines, it is easily observed that each problem in P is also member of the class NP.

The question of whether P equals NP is one of the most important open questions in theoretical computer science because of the wide implications of a solution. If the answer is yes, many important problems can be shown to have more efficient solutions. These include various types of integer programming problems in operations research, many problems in logistics, protein structure prediction in biology, and the ability to find formal proofs of pure mathematics theorems. The P versus NP problem is one of the Millennium Prize Problems proposed by the Clay Mathematics Institute. There is a US$1,000,000 prize for resolving the problem.

Problems in NP not known to be in P or NP-complete

It was shown by Ladner that if P ≠ NP then there exist problems in NP that are neither in P nor NP-complete. Such problems are called NP-intermediate problems. The graph isomorphism problem, the discrete logarithm problem and the integer factorization problem are examples of problems believed to be NP-intermediate. They are some of the very few NP problems not known to be in P or to be NP-complete.

The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic. An important unsolved problem in complexity theory is whether the graph isomorphism problem is in P, NP-complete, or NP-intermediate. The answer is not known, but it is believed that the problem is at least not NP-complete. If graph isomorphism is NP-complete, the polynomial time hierarchy collapses to its second level. Since it is widely believed that the polynomial hierarchy does not collapse to any finite level, it is believed that graph isomorphism is not NP-complete. The best algorithm for this problem, due to Laszlo Babai and Eugene Luks has run time $2^{O(\sqrt{}(n \log n))}$ for graphs with n vertices.

The integer factorization problem is the computational problem of determining the prime factorization of a given integer. Phrased as a decision problem, it is the problem of deciding whether the input has a factor less than k. No efficient integer factorization algorithm is known, and this fact forms the basis of several modern cryptographic systems, such as the RSA algorithm. The integer factorization problem is in NP and in co-NP (and even in UP and co-UP). If the problem is NP-complete, the polynomial time hierarchy will collapse to its first level (i.e., NP will equal co-NP). The best known algorithm for integer factorization is the general number field sieve, which takes time $O(e^{(64/9)1/3(n.\log 2)1/3(\log (n.\log 2))2/3})$ to factor an n-bit integer. However, the best known quantum algorithm for this problem, Shor's algorithm, does run in polynomial time. Unfortunately, this fact doesn't say much about where the problem lies with respect to non-quantum complexity classes.

Separations between other complexity classes

Many known complexity classes are suspected to be unequal, but this has not been proved. For instance P ⊆ NP ⊆ PP ⊆ PSPACE, but it is possible that P = PSPACE. If P is not equal to NP, then P is not equal to PSPACE either. Since there are many known complexity classes between P and PSPACE, such as RP, BPP, PP, BQP, MA, PH, etc., it is possible that all these complexity classes collapse to one class. Proving that any of these classes are unequal would be a major breakthrough in complexity theory.

Along the same lines, co-NP is the class containing the complement problems (i.e. problems with the *yes*/*no* answers reversed) of NP problems. It is believed that NP is not equal to co-NP; however, it has not yet been proven. It has been shown that if these two complexity classes are not equal then P is not equal to NP.

Similarly, it is not known if L (the set of all problems that can be solved in logarithmic space) is strictly contained in P or equal to P. Again, there are many complexity classes between the two, such as NL and NC, and it is not known if they are distinct or equal classes.

Intractability

Problems that can be solved but not fast enough for the solution to be useful are called *intractable*. In complexity theory, problems that lack polynomial-time solutions are considered to be intractable for more than the smallest inputs. In fact, the Cobham–Edmonds thesis states that only those problems that can be solved in polynomial time can be feasibly computed on some computational device. Problems that are known to be intractable in this sense include those that are EXPTIME-hard. If NP is not the same as P, then the NP-complete problems are also intractable in this sense. To see why exponential-time algorithms might be unusable in practice, consider a program that makes 2^n operations before halting. For small n, say 100, and assuming for the sake of example that the computer does 10^{12} operations each second, the program would run for about 4×10^{10} years, which is roughly the age of the universe. Even with a much faster computer, the program would only be useful for very small instances and in that sense the intractability of a problem is somewhat independent of technological progress. Nevertheless a polynomial time algorithm is not always practical. If its running time is, say, n^{15}, it is unreasonable to consider it efficient and it is still useless except on small instances.

What intractability means in practice is open to debate. Saying that a problem is not in P does not imply that all large cases of the problem are hard or even that most of them are. For example the decision problem in Presburger arithmetic has been shown not to be in P, yet algorithms have been written that solve the problem in reasonable times in most cases. Similarly, algorithms can solve the NP-complete knapsack problem over a wide range of sizes in less than quadratic time and SAT solvers routinely handle large instances of the NP-complete Boolean satisfiability problem.

Continuous complexity theory

Continuous complexity theory can refer to complexity theory of problems that involve continuous functions that are approximated by discretizations, as studied in numerical analysis. One approach to complexity theory of numerical analysis is information based complexity.

Continuous complexity theory can also refer to complexity theory of the use of analog computation, which uses continuous dynamical systems and differential equations. Control theory can be considered a form of computation and differential equations are used in the modelling of continuous-time and hybrid discrete-continuous-time systems.

History

Before the actual research explicitly devoted to the complexity of algorithmic problems started off, numerous foundations were laid out by various researchers. Most influential among these was the definition of Turing machines by Alan Turing in 1936, which turned out to be a very robust and flexible notion of computer.

Fortnow & Homer (2003) date the beginning of systematic studies in computational complexity to the seminal paper "On the Computational Complexity of Algorithms" by Juris Hartmanis and Richard Stearns (1965), which laid out the definitions of time and space complexity and proved the hierarchy theorems.

According to Fortnow & Homer (2003), earlier papers studying problems solvable by Turing machines with specific bounded resources include John Myhill's definition of linear bounded automata (Myhill 1960), Raymond Smullyan's study of rudimentary sets (1961), as well as Hisao Yamada's paper on real-time computations (1962). Somewhat earlier, Boris Trakhtenbrot (1956), a pioneer in the field from the USSR, studied another specific complexity measure. As he remembers:

> “ However, [my] initial interest [in automata theory] was increasingly set aside in favor of computational complexity, an exciting fusion of combinatorial methods, inherited from switching theory, with the conceptual arsenal of the theory of algorithms. These ideas had occurred to me earlier in 1955 when I coined the term "signalizing function", which is nowadays commonly known as "complexity measure". ”

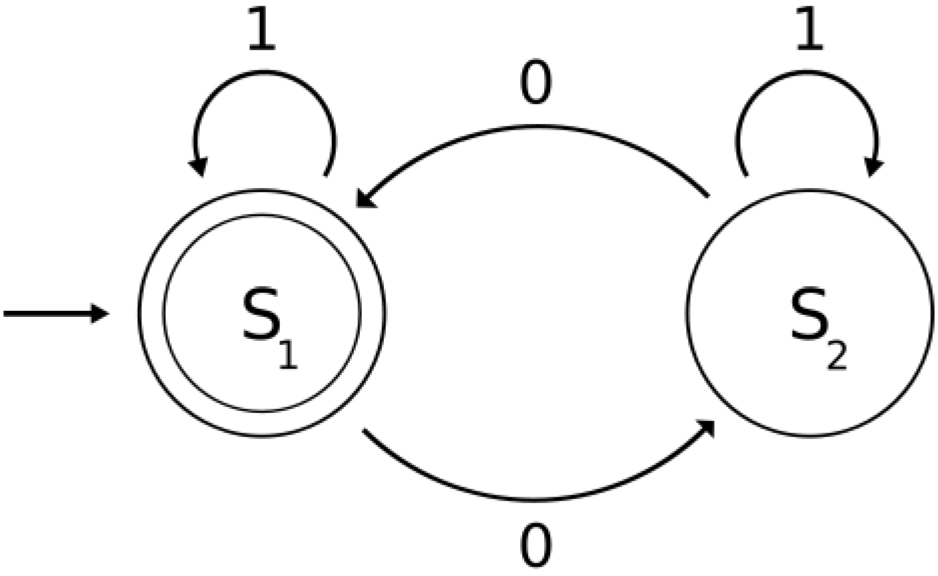
—Boris Trakhtenbrot, From Logic to Theoretical Computer Science – An Update. In: *Pillars of Computer Science*, LNCS 4800, Springer 2008.

In 1967, Manuel Blum developed an axiomatic complexity theory based on his axioms and proved an important result, the so called, speed-up theorem. The field really began to flourish when the US researcher Stephen Cook and, working independently, Leonid Levin in the USSR, proved that there exist practically relevant problems that are NP-complete. In 1972, Richard Karp took this idea a leap forward with his landmark paper, "Reducibility Among Combinatorial Problems", in which he showed that 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are NP-complete.

Chapter 4

Automata Theory and Quantum Computer

Automata theory



An example of automata and study of mathematical properties of such automata is automata theory

In theoretical computer science, **automata theory** is the study of abstract machines and the computational problems that can be solved using these abstract machines. These abstract machines are called automata.

The figure at right illustrates a finite state machine, which is one well-known variety of automaton. This automaton consists of states (represented in the figure by circles), and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its *transition function* (which takes the current state and the recent symbol as its inputs).

Automata theory is also closely related to formal language theory, as the automata are often classified by the class of formal languages they are able to recognize. An automaton can be a finite representation of a formal language that may be an infinite set.

Automata play a major role in compiler design and parsing.

Automata

Following is an introductory definition of one type of automata, which attempts to help one grasp the essential concepts involved in automata theory.

Informal description

An automaton is supposed to *run* on some given sequence or string of *inputs* in discrete time steps. At each time step, an automaton gets one input that is picked up from a set of *symbols* or *letters*, which is called an *alphabet*. At any time, the symbols so far fed to the automaton as input form a finite sequence of symbols, which is called a *word*. An automaton contains a finite set of states. At each instance in time of some run, automaton is *in* one of its states. At each time step when the automaton reads a symbol, it *jumps* or *transits* to next state depending on its current state and on the symbol currently read. This function in terms of the current state and input symbol is called *transition function*. The automaton *reads* the input word one symbol after another in the sequence and transits from state to state according to the transition function, until the word is read completely. Once the input word has been read, the automaton is said to have been *stopped* and the state at which automaton has stopped is called *final state*. Depending on the final state, it's said that the automaton either *accepts* or *rejects* an input word. There is a subset of states of the automaton, which is defined as the set of *accepting states*. If the final state is an accepting state, then the automaton *accepts* the word. Otherwise, the word is *rejected*. The set of all the words accepted by an automaton is called the *language recognized by the automaton*.

Formal definition

Automaton

An **automaton** is represented formally by the 5-tuple **(Q,Σ,δ,q_0,F)**, where:

- Q is a finite set of *states*.
- Σ is a finite set of *symbols*, called the *alphabet* of the automaton.
- δ is the **transition function**, that is, $\delta: Q \times \Sigma \rightarrow Q$.
- q_0 is the *start state*, that is, the state which the automaton is *in* when no input has been processed yet, where $q_0 \in Q$.
- F is a set of states of Q (i.e. $F \subseteq Q$) called **accept states**.

Input word

An automaton reads a finite string of symbols $a_1,a_2,\ldots., a_n$, where $a_i \in \Sigma$, which is called a *input word*. Set of all words is denoted by Σ^*.

Run

A *run* of the automaton on an input word $w = a_1,a_2,...., a_n \in \Sigma^*$, is a sequence of states $q_0,q_1,q_2,...., q_n$, where $q_i \in Q$ such that q_0 is a start state and $q_i = \delta(q_{i-1},a_i)$ for $0 < i \leq n$. In words, at first the automaton is at the start state q_0 and then automaton reads symbols of the input word in sequence. When automaton reads symbol a_i then it jumps to state $q_i = \delta(q_{i-1},a_i)$. q_n said to be the *final state* of the run.

Accepting word

A word $w \in \Sigma^*$ is accepted by the automaton if $q_n \in F$.

Recognized language

An automaton can recognize a formal language. The recognized language $L \subset \Sigma^*$ by an automaton is the set of all the words that are accepted by the automaton.

Recognizable languages

The recognizable languages is the set of languages that are recognized by some automaton. For above definition of automata the recognizable languages are regular languages. For different definitions of automata, the recognizable languages are different.

Variations in definition of automata

Automata are defined to study useful machines under mathematical formalism. So, the definition of an automaton is open to variations according to the "real world machine", which we want to model using the automaton. People have studied many variations of automata. Above, the most standard variant is described, which is called deterministic finite automaton. The following are some popular variations in the definition of different components of automata.

Input

- *Finite input*: An automaton that accepts only finite sequence of words. The above introductory definition only accepts finite words.
- *Infinite input*: An automaton that accepts infinite words (ω-words). Such automata are called *ω-automata*.
- *Tree word input*: The input may be a *tree of symbols* instead of sequence of symbols. In this case after reading each symbol, the automaton *reads* all the successor symbols in the input tree. It is said that the automaton *makes one copy* of itself for each successor and each such copy starts running on one of the successor symbol from the state according to the transition relation of the automaton. Such an automaton is called tree automaton.

States

- *Finite states*: An automaton that contains only a finite number of states. The above introductory definition describes automata with finite numbers of states.

- *Infinite states*: An automaton that may not have a finite number of states, or even a countable number of states. For example, the quantum finite automaton or topological automaton has uncountable infinity of states.
- *Stack memory*: An automaton may also contain some extra memory in the form of a stack in which symbols can be pushed and popped. This kind of automaton is called a *pushdown automaton*

Transition function

- *Deterministic*: For a given current state and an input symbol, if an automaton can only jump to one and only one state then it is a *deterministic automaton*.
- *Nondeterministic*: An automaton that, after reading an input symbol, may jump into any of a number of states, as licensed by its transition relation. Notice that the term transition function is replaced by transition relation: The automaton *non-deterministically* decides to jump into one of the allowed choices. Such automaton are called *nondeterministic automaton*.

- *Alternation*: This idea is quite similar to tree automaton, but orthogonal. The automaton may run its *multiple copies* on the *same* next read symbol. Such automata are called *alternating automaton*. Acceptance condition must satisfy all runs of such *copies* to accept the input.

Acceptance condition

- *Acceptance of finite words*: Same as described in the informal definition above.
- *Acceptance of infinite words*: an *omega automaton* cannot have final states, as infinite words never terminate. Rather, acceptance of the word is decided by looking at the infinite sequence of visited states during the run.
- *Probabilistic acceptance*: An automaton need not strictly accept or reject an input. It may accept the input with some probability between zero and one. For example, quantum finite automaton, geometric automaton and *metric automaton* has probabilistic acceptance.

Different combinations of the above variations produce many variety of automaton.

Automata theory

Automata theory is a subject matter which studies properties of various types of automata. For example, following questions are studied about a given type of automata.

- Which class of formal languages is recognizable by some type of automata?(Recognizable languages)
- Is certain automata *closed* under union, intersection, or complementation of formal languages?(Closure properties)

- How much is a type of automata expressive in terms of recognizing class of formal languages? And, their relative expressive power?(Language Hierarchy)

Automata theory also studies if there exist any effective algorithm or not to solve problems similar to following list.

- Does an automaton accept any input word?(emptiness checking)
- Is it possible to transform a given non-deterministic automaton into deterministic automaton without changing the recognizing language?(Determinization)
- For a given formal language, what is the smallest automaton that recognize it?(Minimization).

Classes of automata

Following is an incomplete list of some types of automata.

Automata	**Recognizable language**
Deterministic finite automata (DFA)	regular languages
Nondeterministic finite automata (NFA)	regular languages
Nondeterministic finite automata with ε transitions (FND-ε or ε-NFA)	regular languages
Pushdown automata (PDA)	context-free languages
Linear bounded automata (LBA)	context-sensitive language
Turing machines	recursively enumerable languages
Timed automata	
Deterministic Büchi automata	omega limit languages
Nondeterministic Büchi automata	omega regular languages
Nondeterministic/Deterministic Rabin automata	omega regular languages
Nondeterministic/Deterministic Streett automata	omega regular languages
Nondeterministic/Deterministic parity automata	omega regular languages
Nondeterministic/Deterministic Muller automata	omega regular languages

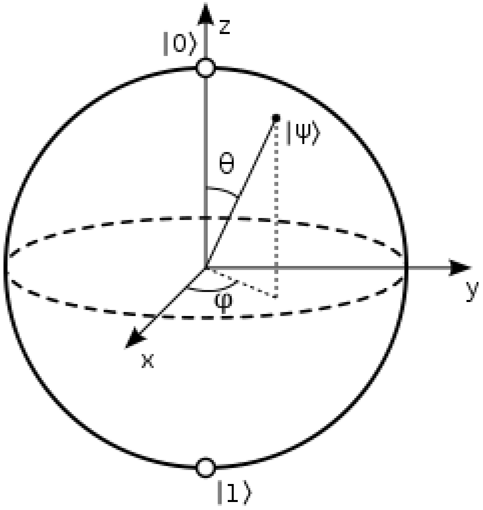
Discrete, continuous, and hybrid automata

Normally automata theory describes the states of abstract machines but there are analog automata or continuous automata or hybrid discrete-continuous automata, using analog data, continuous time, or both.

Applications

Each model in automata theory play varied roles in several applied areas. Finite automata is used in text processing, compilers, and hardware design. Context-free grammar is used in programming languages and artificial intelligence. Originally, CFG were used in the study of the human languages. Cellular automata is used in the field of biology, the most common example being John Conway's Game of Life. Some other examples which could be explained using automata theory in biology include mollusk and pine cones growth and pigmentation patterns. Going further, Stephen Wolfram claims that the entire universe could be explained by machines with a finite set of states and rules with a single initial condition. Other areas of interest which he has related to automata theory include: fluid flow, snowflake and crystal formation, chaos theory, cosmology, and financial analysis.

Quantum computer



The Bloch sphere is a representation of a qubit, the fundamental building block of quantum computers.

A **quantum computer** is a device for computation that makes direct use of quantum mechanical phenomena, such as superposition and entanglement, to perform operations on data. Quantum computers are different from traditional computers based on transistors. The basic principle behind quantum computation is that quantum properties can be used to represent data and perform operations on these data. A theoretical model is the quantum Turing machine, also known as the universal quantum computer.

Although quantum computing is still in its infancy, experiments have been carried out in which quantum computational operations were executed on a very small number of qubits (quantum bit). Both practical and theoretical research continues, and many national government and military funding agencies support quantum computing research to develop quantum computers for both civilian and national security purposes, such as cryptanalysis.

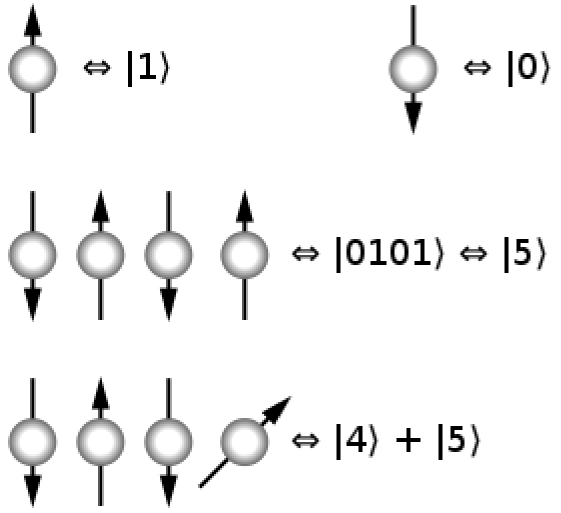
If large-scale quantum computers can be built, they will be able to solve certain problems much faster than any current classical computers (for example Shor's algorithm). Quantum computers do not allow the computation of functions that are not theoretically computable by classical computers, i.e. they do not alter the Church–Turing thesis. The gain is only in efficiency.

Basis

A classical computer has a memory made up of bits, where each bit represents either a one or a zero. A quantum computer maintains a sequence of qubits. A single qubit can represent a one, a zero, or, crucially, any quantum superposition of these; moreover, a pair of qubits can be in any quantum superposition of 4 states, and three qubits in any superposition of 8. In general a quantum computer with n qubits can be in an arbitrary superposition of up to 2^n different states simultaneously (this compares to a normal computer that can only be in *one* of these 2^n states at any one time). A quantum computer operates by manipulating those qubits with a fixed sequence of quantum logic gates. The sequence of gates to be applied is called a quantum algorithm.

An example of an implementation of qubits for a quantum computer could start with the use of particles with two spin states: "down" and "up" (typically written $|{\downarrow}\rangle$and $|{\uparrow}\rangle$, or $|0\rangle$and $|1\rangle$). But in fact any system possessing an observable quantity A which is *conserved* under time evolution and such that A has at least two discrete and sufficiently spaced consecutive eigenvalues, is a suitable candidate for implementing a qubit. This is true because any such system can be mapped onto an effective spin-1/2 system.

Bits vs. qubits



qubits can be in a superposition of all the clasically allowed states

Qubits are made up of controlled particles and the means of control (e.g. devices that trap particles and switch them from one state to another).

Consider first a classical computer that operates on a three-bit register. The state of the computer at any time is a probability distribution over the $2^3 = 8$ different three-bit strings 000, 001, 010, 011, 100, 101, 110, 111. If it is a deterministic computer, then it is in exactly one of these states with probability 1. However, if it is a probabilistic computer, then there is a possibility of it being in any *one* of a number of different states. We can describe this probabilistic state by eight nonnegative numbers *a*,*b*,*c*,*d*,*e*,*f*,*g*,*h* (where *a* = probability computer is in state 000, *b* = probability computer is in state 001, etc.). There is a restriction that these probabilities sum to 1.

The state of a three-qubit quantum computer is similarly described by an eight-dimensional vector (*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*), called a ket. However, instead of adding to one, the sum of the *squares* of the coefficient magnitudes, $| a |^2 + | b |^2 + ... + | h |^2$, must equal one. Moreover, the coefficients are complex numbers. Since states are represented by complex wavefunctions, two states being added together will undergo interference. This is a key difference between quantum computing and probabilistic classical computing.

If you measure the three qubits, then you will observe a three-bit string. The probability of measuring a string will equal the squared magnitude of that string's coefficients (using our example, probability that we read state as 000 = $| a |^2$, probability that we read state as 001 = $| b |^2$, etc..). Thus a measurement of the quantum state with coefficients (*a*,*b*,...,*h*) gives the classical probability distribution ($| a |^2, | b |^2,..., | h |^2$). We say that the quantum state "collapses" to a classical state.

Note that an eight-dimensional vector can be specified in many different ways, depending on what basis you choose for the space. The basis of three-bit strings 000, 001, ..., 111 is known as the computational basis, and is often convenient, but other bases of unit-length, orthogonal vectors can also be used. Ket notation is often used to make explicit the choice of basis. For example, the state (*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*) in the computational basis can be written as

$a\,|000\rangle + b\,|001\rangle + c\,|010\rangle + d\,|011\rangle + e\,|100\rangle + f\,|101\rangle + g\,|110\rangle + h\,|111\rangle$, where, e.g., $|010\rangle$= (0,0,1,0,0,0,0,0).

The computational basis for a single qubit (two dimensions) is $|0\rangle$= (1,0), $|1\rangle$= (0,1), but another common basis are the eigenvectors of the Pauli-x operator: $|+\rangle = \frac{1}{\sqrt{2}}\,(1,1)$ and $|-\rangle = \frac{1}{\sqrt{2}}\,(1,-1)$.

Note that although recording a classical state of *n* bits, a 2^n-dimensional probability distribution, requires an exponential number of real numbers, practically we can always think of the system as being exactly one of the *n*-bit strings—we just don't know which one. Quantum mechanically, this is not the case, and all 2^n complex coefficients need to be kept track of to see how the quantum system evolves. For example, a 300-qubit quantum computer has a state described by 2^{300} (approximately 10^{90}) complex numbers, more than the number of atoms in the observable universe.

Operation

While a classical three-bit state and a quantum three-qubit state are both eight-dimensional vectors, they are manipulated quite differently for classical or quantum computation. For computing in either case, the system must be initialized, for example into the all-zeros string, $|000\rangle$, corresponding to the vector (1,0,0,0,0,0,0,0). In classical randomized computation, the system evolves according to the application of stochastic matrices, which preserve that the probabilities add up to one (i.e., preserve the L1 norm). In quantum computation, on the other hand, allowed operations are unitary matrices, which are effectively rotations (they preserve that the sum of the squares add up to one, the Euclidean or L2 norm). (Exactly what unitaries can be applied depend on the physics of the quantum device.) Consequently, since rotations can be undone by rotating backward, quantum computations are reversible. (Technically, quantum operations can be probabilistic combinations of unitaries, so quantum computation really does generalize classical computation.)

Finally, upon termination of the algorithm, the result needs to be read off. In the case of a classical computer, we *sample* from the probability distribution on the three-bit register to obtain one definite three-bit string, say 000. Quantum mechanically, we *measure* the three-qubit state, which is equivalent to collapsing the quantum state down to a classical distribution (with the coefficients in the classical state being the squared magnitudes of the coefficients for the quantum state, as described above) followed by sampling from that distribution. Note that this destroys the original quantum state. Many algorithms will only give the correct answer with a certain probability, however by repeatedly initializing, running and measuring the quantum computer, the probability of getting the correct answer can be increased.

Potential

Integer factorization is believed to be computationally infeasible with an ordinary computer for large integers if they are the product of few prime numbers (e.g., products of two 300-digit primes). By comparison, a quantum computer could efficiently solve this problem using Shor's algorithm to find its factors. This ability would allow a quantum computer to decrypt many of the cryptographic systems in use today, in the sense that there would be a polynomial time (in the number of digits of the integer) algorithm for solving the problem. In particular, most of the popular public key ciphers are based on the difficulty of factoring integers (or the related discrete logarithm problem which can also be solved by Shor's algorithm), including forms of RSA. These are used to protect secure Web pages, encrypted email, and many other types of data. Breaking these would have significant ramifications for electronic privacy and security.

However, other existing cryptographic algorithms don't appear to be broken by these algorithms. Some public-key algorithms are based on problems other than the integer factorization and discrete logarithm problems to which Shor's algorithm applies, like the McEliece cryptosystem based on a problem in coding theory. Lattice based cryptosystems are also not known to be broken by quantum computers, and finding a polynomial time algorithm for solving the dihedral hidden subgroup problem, which would break many lattice based cryptosystems, is a well-studied open problem. It has been proven that applying Grover's algorithm to break a symmetric (secret key) algorithm by brute force requires roughly $2^{n/2}$ invocations of the underlying cryptographic algorithm, compared with roughly 2^n in the classical case, meaning that symmetric key lengths are effectively halved: AES-256 would have the same security against an attack using Grover's algorithm that AES-128 has against classical brute-force search. Quantum cryptography could potentially fulfill some of the functions of public key cryptography.

Besides factorization and discrete logarithms, quantum algorithms offering a more than polynomial speedup over the best known classical algorithm have been found for several problems, including the simulation of quantum physical processes from chemistry and solid state physics, the approximation of Jones polynomials, and solving Pell's equation. No mathematical proof has been found that shows that an equally fast classical algorithm cannot be discovered, although this is considered unlikely. For some problems, quantum computers offer a polynomial speedup. The most well-known example of this is *quantum database search*, which can be solved by Grover's algorithm using quadratically fewer queries to the database than are required by classical algorithms. In this case the advantage is provable. Several other examples of provable quantum speedups for query problems have subsequently been discovered, such as for finding collisions in two-to-one functions and evaluating NAND trees.

Consider a problem that has these four properties:

1. The only way to solve it is to guess answers repeatedly and check them,
2. There are *n* possible answers to check,
3. Every possible answer takes the same amount of time to check, and
4. There are no clues about which answers might be better: generating possibilities randomly is just as good as checking them in some special order.

An example of this is a password cracker that attempts to guess the password for an encrypted file (assuming that the password has a maximum possible length).

For problems with all four properties, the time for a quantum computer to solve this will be proportional to the square root of *n*. That can be a very large speedup, reducing some problems from years to seconds. It can be used to attack symmetric ciphers such as Triple DES and AES by attempting to guess the secret key.

Grover's algorithm can also be used to obtain a quadratic speed-up [over a brute-force search] for a class of problems known as NP-complete.

Since chemistry and nanotechnology rely on understanding quantum systems, and such systems are impossible to simulate in an efficient manner classically, many believe quantum simulation will be one of the most important applications of quantum computing.

There are a number of practical difficulties in building a quantum computer, and thus far quantum computers have only solved trivial problems. David DiVincenzo, of IBM, listed the following requirements for a practical quantum computer:

- scalable physically to increase the number of qubits;
- qubits can be initialized to arbitrary values;
- quantum gates faster than decoherence time;
- universal gate set;
- qubits can be read easily.

Quantum decoherence

One of the greatest challenges is controlling or removing quantum decoherence. This usually means isolating the system from its environment as the slightest interaction with the external world would cause the system to decohere. This effect is irreversible, as it is non-unitary, and is usually something that should be highly controlled, if not avoided. Decoherence times for candidate systems, in particular the transverse relaxation time T_2 (for NMR and MRI technology, also called the *dephasing time*), typically range between nanoseconds and seconds at low temperature.

These issues are more difficult for optical approaches as the timescales are orders of magnitude shorter and an often-cited approach to overcoming them is optical pulse shaping. Error rates are typically proportional to the ratio of operating time to decoherence time, hence any operation must be completed much more quickly than the decoherence time.

If the error rate is small enough, it is thought to be possible to use quantum error correction, which corrects errors due to decoherence, thereby allowing the total calculation time to be longer than the decoherence time. An often cited figure for required error rate in each gate is 10^{-4}. This implies that each gate must be able to perform its task in one 10,000th of the decoherence time of the system.

Meeting this scalability condition is possible for a wide range of systems. However, the use of error correction brings with it the cost of a greatly increased number of required qubits. The number required to factor integers using Shor's algorithm is still polynomial, and thought to be between L and L^2, where L is the number of bits in the number to be factored; error correction algorithms would inflate this figure by an additional factor of L. For a 1000-bit number, this implies a need for about 10^4 qubits without error correction. With error correction, the figure would rise to about 10^7 qubits. Note that computation time is about L^2 or about 10^7 steps and on 1 MHz, about 10 seconds.

A very different approach to the stability-decoherence problem is to create a topological quantum computer with anyons, quasi-particles used as threads and relying on braid theory to form stable logic gates.

Developments

There are a number of quantum computing candidates, among those:

- Superconductor-based quantum computers (including SQUID-based quantum computers)
- Trapped ion quantum computer
- Optical lattices
- Topological quantum computer
- Quantum dot on surface (e.g. the Loss-DiVincenzo quantum computer)
- Nuclear magnetic resonance on molecules in solution (liquid NMR)
- Solid state NMR Kane quantum computers
- Electrons on helium quantum computers
- Cavity quantum electrodynamics (CQED)
- Molecular magnet
- Fullerene-based ESR quantum computer
- Optic-based quantum computers (Quantum optics)
- Diamond-based quantum computer
- Bose–Einstein condensate-based quantum computer
- Transistor-based quantum computer - string quantum computers with entrainment of positive holes using an electrostatic trap
- Spin-based quantum computer
- Adiabatic quantum computation

- Rare-earth-metal-ion-doped inorganic crystal based quantum computers

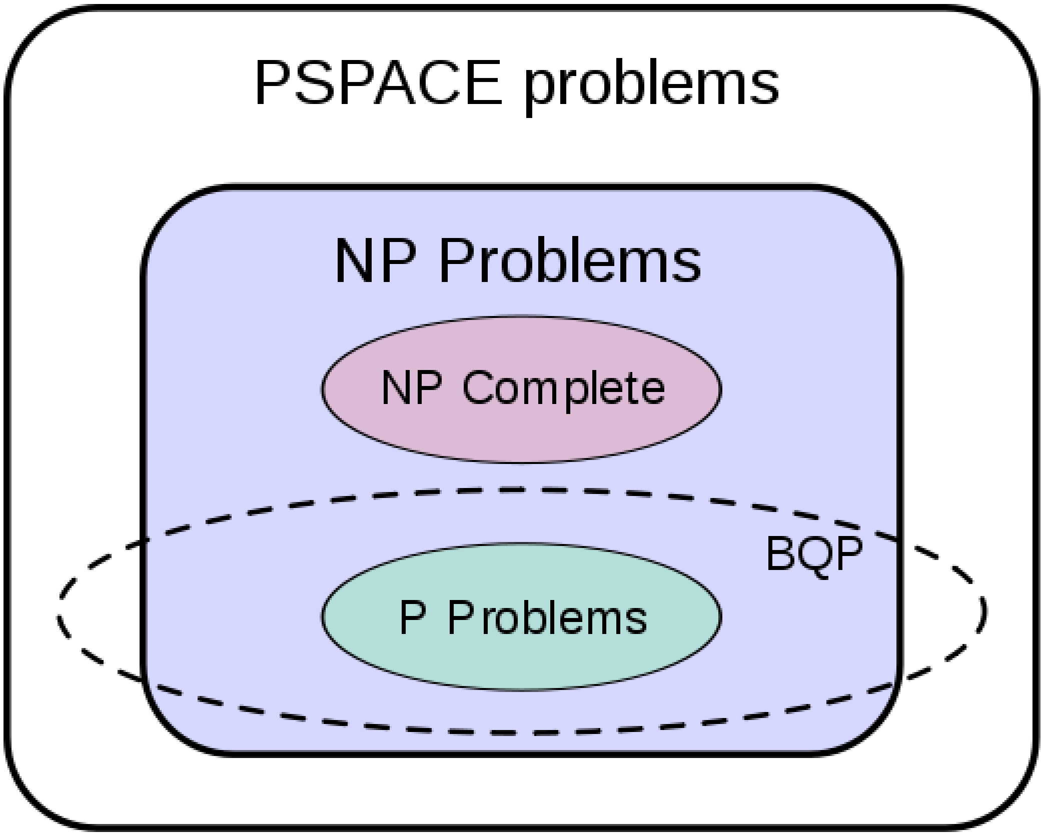
The large number of candidates demonstrates that the topic, in spite of rapid progress, is still in its infancy. But at the same time there is also a vast amount of flexibility.

In 2005, researchers at the University of Michigan built a semiconductor chip which functioned as an ion trap. Such devices, produced by standard lithography techniques, may point the way to scalable quantum computing tools. An improved version was made in 2006.

In 2009, researchers at Yale University created the first rudimentary solid-state quantum processor. The two-qubit superconducting chip was able to run elementary algorithms. Each of the two artificial atoms (or qubits) were made up of a billion aluminum atoms but they acted like a single one that could occupy two different energy states.

Another team, working at the University of Bristol, also created a silicon-based quantum computing chip, based on quantum optics. The team was able to run Shor's algorithm on the chip. The latest developments [for 2010] can be found in .

Relation to computational complexity theory



The suspected relationship of BQP to other problem spaces.

The class of problems that can be efficiently solved by quantum computers is called BQP, for "bounded error, quantum, polynomial time". Quantum computers only run probabilistic algorithms, so BQP on quantum computers is the counterpart of BPP ("bounded error, probabilistic, polynomial time") on classical computers. It is defined as the set of problems solvable with a polynomial-time algorithm, whose probability of error is bounded away from one half. A quantum computer is said to "solve" a problem if, for every instance, its answer will be right with high probability. If that solution runs in polynomial time, then that problem is in BQP.

BQP is contained in the complexity class *#P* (or more precisely in the associated class of decision problems $P^{\#P}$), which is a subclass of PSPACE.

BQP is suspected to be disjoint from NP-complete and a strict superset of P, but that is not known. Both integer factorization and discrete log are in BQP. Both of these problems are NP problems suspected to be outside BPP, and hence outside P. Both are suspected to not be NP-complete. There is a common misconception that quantum computers can solve NP-complete problems in polynomial time. That is not known to be true, and is generally suspected to be false.

Although quantum computers may be faster than classical computers, those described above can't solve any problems that classical computers can't solve, given enough time and memory (however, those amounts might be practically infeasible). A Turing machine can simulate these quantum computers, so such a quantum computer could never solve an undecidable problem like the halting problem. The existence of "standard" quantum computers does not disprove the Church–Turing thesis. It has been speculated that theories of quantum gravity, such as M-theory or loop quantum gravity, may allow even faster computers to be built. Currently, it's an open problem to even *define* computation in such theories due to the problem of time, i.e. there's no obvious way to describe what it means for an observer to submit input to a computer and later receive output.

Chapter 5

Models of computation

1. Lambda calculus

In mathematical logic and computer science, **lambda calculus**, also written as **λ-calculus**, is a formal system for function definition, function application and recursion. The portion of lambda calculus relevant to computation is now called the **untyped lambda calculus**. In both typed and untyped versions, ideas from lambda calculus have found application in the fields of logic, recursion theory (computability), and linguistics, and have played an important role in the development of the theory of programming languages (with untyped lambda calculus being the original inspiration for functional programming, in particular Lisp, and typed lambda calculi serving as the foundation for modern type systems).

History

Lambda calculus was introduced by Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics. The original system was shown to be logically inconsistent in 1935 when Stephen Kleene and J. B. Rosser developed the Kleene–Rosser paradox.

Subsequently, in 1936 Church isolated and published just the portion relevant to computation, what is now called the untyped lambda calculus. In 1940, he also introduced a computationally weaker but logically consistent system, known as the simply typed lambda calculus.

Informal description

Motivation

Functions are a fundamental concept within computer science and mathematics. The λ-calculus provides simple semantics for computation with functions so that properties of functional computation can be studied.

Consider the following two examples. The identity function `I(x) = x` takes a single input, `x`, and immediately returns `x` (i.e. the identity does nothing with its input), whereas the function `sqadd(x, y) = x*x + y*y` takes a pair of inputs, `x` and `y` and returns the sum of their squares, `x*x + y*y`. Using these two examples, we can make some useful observations that motivate the major ideas in the lambda calculus.

The first observation is that functions need not be explicitly named. That is, the function `sqadd(x, y) = x*x + y*y` can be rewritten in *anonymous form* as `(x, y) ↦ x*x + y*y`, (read as "the pair of `x` and `y` is mapped to `x*x + y*y`"). Similarly, `I(x) = x` can be rewritten in anonymous form as `x ↦ x`, where the input is simply mapped to itself.

The second observation is that the specific choice of name for a function's arguments is largely irrelevant. That is, `x ↦ x` and `y ↦ y` express the same function: the identity. Similarly, `(x, y) ↦ x*x + y*y` and `(u, v) ↦ u*u + v*v` also express the same function.

Finally, any function that requires two inputs, for instance the before mentioned `sqadd` function, can be reworked into an equivalent function that accepts a single input, and as output returns *another* function, that in turn accepts a single input. For example, `(x, y) ↦ x*x + y*y` can be reworked into `x ↦ (y ↦ x*x + y*y)`. This transformation is called currying, and can be generalized to functions accepting an arbitrary number of arguments.

Currying may be best grasped intuitively through the use of an example. Compare the function `(x, y) ↦ x*x + y*y` with its curried form, `x ↦ (y ↦ x*x + y*y)`. Given two arguments, we have:

```
((x, y) ↦ x*x + y*y)(5, 2)
= 5*5 + 2*2 = 29.
```

However, using currying, we have:

```
((x ↦ (y ↦ x*x + y*y))(5))(2)
= (y ↦ 5*5 + y*y)(2)
= 5*5 + 2*2 = 29
```

and we see the uncurried and curried forms compute the same result. Notice that x*x has become a constant.

The lambda calculus

The lambda calculus consists of **lambda terms** and some extra operations.

Since the names of functions are largely a convenience, the lambda calculus has no means of naming a function. Since all functions expecting more than one input can be transformed into equivalent functions accepting a single input (via Currying), the lambda calculus has no means for creating a function that accepts more than one argument. Since the names of arguments are largely irrelevant, the native notion of equality on lambda terms is **alpha-equivalence** (see below), which codifies this principle.

Lambda terms

The syntax of lambda terms is particularly simple. There are three ways in which to obtain them:

- a lambda term may be a variable, `x`;
- if `t` is a lambda term, and `x` is a variable, then `λx.t` is a lambda term (called a **lambda abstraction**);
- if `t` and `s` are lambda terms, then `ts` is a lambda term (called an **application**).

Nothing else is a lambda term, though bracketing may be used to disambiguate terms.

Intuitively, a lambda abstraction `λx.t` represents an anonymous function that takes a single input, and the `λ` is said to **bind** `x` in `t`, and an application `ts` represents the application of input `s` to some function `t`. In the lambda calculus, functions are taken to be **first class values**, so functions may be used as the inputs to other functions, and functions may return functions as their outputs.

For example, `λx.x` represents the identity function, `x ↦ x`, and `(λx.x)y` represents the identity function applied to `y`. Further, `(λx.y)` represents the **constant function** `x ↦ y`, the function that always returns y, no matter the input. It should be noted that function application is left-associative, so `(λx.x)y z = ((λx.x)y)z`.

Lambda terms on their own aren't particularly interesting. What makes them interesting are the various notions of **equivalence** and **reduction** that can be defined over them.

Alpha equivalence

A basic form of equivalence, definable on lambda terms, is alpha equivalence. This states that the particular choice of bound variable, in a lambda abstraction, doesn't (usually) matter. For instance, `λx.x` and `λy.y` are alpha-equivalent lambda terms, representing the same identity function. Note that the terms `x` and `y` **aren't** alpha-equivalent, because they are not bound in a lambda abstraction. In many presentations, it is usual to identify alpha-equivalent lambda terms.

The following definitions are necessary in order to be able to define beta reduction.

Free variables

The **free variables** of a term are those variables not bound by a lambda abstraction. That is, the free variables of `x` are just `x`; the free variables of `λx.t` are the free variables of `t`, with `x` removed, and the free variables of `ts` are the union of the free variables of `t` and `s`.

For example, the lambda term representing the identity `λx.x` has no free variables, but the constant function `λx.y` has a single free variable, `y`.

Capture-avoiding substitutions

Using the definition of free variables, we may now define a capture-avoiding substitution. Suppose `t`, `s` and `r` are lambda terms and `x` and `y` are variables where `x` does not equal `y`. We write `t[x := r]` for the substitution of `r` for `x` in `t`, in a capture-avoiding manner. That is:

- `x[x := r] = r`;
- `y[x := r] = y`;
- `(ts)[x := r] = (t[x := r])(s[x := r])`;
- `(λx.t)[x := r] = λx.t`;
- `(λy.t)[x := r] = λy.(t[x := r])` if `x ≠ y` and `y` is not in the free variables of `r` (sometimes said "`y` is fresh for `r`").

For example, `(λx.x)[y := y] = λx.(x[y := y]) = λx.x`, and `((λx.y)x)[x := y] = ((λx.y)[x := y])(x[x := y]) = (λx.y)y`.

The freshness condition (requiring that `y` is not in the free variables of `r`) is crucial in order to ensure that substitution does not change the meaning of functions. For example, suppose we define another substitution action without the freshness condition. Then, `(λx.y)[y := x] = λx.(y[y := x]) = λx.x`, and the constant function `λx.y` turns into the identity `λx.x` by substitution.

If our freshness condition is not met, then we may simply alpha-rename with a suitably fresh variable. For example, switching back to our correct notion of substitution, in `(λx.y)[y := x]` the lambda abstraction can be renamed with a fresh variable `z`, to obtain `(λz.y)[y := x] = λz.(y[y := x]) = λz.x`, and the meaning of the function is preserved by substitution.

Beta reduction

Beta reduction states that an application of the form `(λx.t)s` reduces to the term `t[x := s]` (we write `(λx.t)s → t[x := s]` as a convenient shorthand for "`(λx.t)s` beta reduces to `t[x := s]`"). For example, for every `s` we have `(λx.x)s → x[x := s] = s`, demonstrating that `λx.x` really is the identity. Similarly, `(λx.y)s → y[x := s] = y`, demonstrating that `λx.y` really is a constant function.

The lambda calculus may be seen as an idealised functional programming language, like Haskell or Standard ML. Under this view, beta reduction corresponds to a computational step, and in the untyped lambda calculus, as presented here, reduction need not terminate. For instance, consider the term `(λx.xx)(λx.xx)`. Here, we have `(λx.xx)(λx.xx) → (xx)[x := λx.xx] = (x[x := λx.xx])(x[x := λx.xx]) = (λx.xx)(λx.xx)`. That is, the term reduces to itself in a single beta reduction, and therefore reduction will never terminate.

Another problem with the untyped lambda calculus is the inability to distinguish between different kinds of data. For instance, we may want to write a function that only operates on numbers. However, in the untyped lambda calculus, there's no way to prevent our function from being applied to truth values, or strings, for instance.

Typed lambda calculi, that will be introduced later, attempt to rule out as many misbehaved terms as possible.

Formal definition

Definition

Lambda expressions are composed of

variables $v_1, v_2, ..., v_n$
the abstraction symbols λ and .
parentheses ( )

The set of lambda expressions, Λ, can be defined recursively:

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$
3. If $M, N \in \Lambda$, then $(M\ N) \in \Lambda$

Instances of rule 2 are known as abstractions and instances of rule 3 are known as applications.

Notation

To keep the notation of lambda expressions uncluttered, the following conventions are usually applied.

- Outermost parentheses are dropped: M N instead of (M N).
- Applications are assumed to be left associative: M N P means (M N) P.
- The body of an abstraction extends as far right as possible: λx.M N means λx.(M N) and not (λx.M) N.
- A sequence of abstractions is contracted: λx.λy.λz.N is abbreviated as λxyz.N.

Free and bound variables

The abstraction operator, λ, is said to bind its variable wherever it occurs in the body of the abstraction. Variables that fall within the scope of a lambda are said to be *bound*. All other variables are called *free*. For example in the following expression y is a bound variable and x is free: `λy.x x y`. Also note that a variable binds to its "nearest" lambda. In the following expression one single occurrence of x is bound by the second lambda: `λx.y (λx.z x)`

The set of *free variables* of a lambda expression, M, is denoted as FV(M) and is defined by recursion on the structure of the terms, as follows:

1. FV(x) = {x}, where x is a variable
2. FV(λx.M) = FV(M) - {x}
3. FV(M N) = FV(M) ∪ FV(N)

An expression which contains no free variables is said to be *closed*. Closed lambda expressions are also known as combinators and are equivalent to terms in combinatory logic.

Reduction

The meaning of lambda expressions is defined by how expressions can be reduced.

There are three kinds of reduction:

- **α-conversion**: changing bound variables;
- **β-reduction**: applying functions to their arguments;
- **η-conversion**: which captures a notion of extensionality.

We also speak of the resulting equivalences: two expressions are *β-equivalent* if they can be β-converted into the same expression, and α/η-equivalence are defined similarly.

The term *redex*, short for *reducible expression*, refers to subterms which can be reduced by one of the reduction rules. For example, `(λx.M) N` is a beta-redex; if `x` is not free in `M`, `λx.M x` is an eta-redex. The expression to which a redex reduces is called its reduct; using the previous example, the reducts of these expressions are respectively `M[x:=N]` and `M`.

α-conversion

Alpha-conversion, sometimes known as alpha-renaming, allows bound variable names to be changed. For example, alpha-conversion of `λx.x` might yield `λy.y`. Terms that differ only by alpha-conversion are called *α-equivalent*. Frequently in uses of lambda calculus, α-equivalent terms are considered to be equivalent.

The precise rules for alpha-conversion are not completely trivial. First, when alpha-converting an abstraction, the only variable occurrences that are renamed are those that are bound to the same abstraction. For example, an alpha-conversion of `λx.λx.x` could result in `λy.λx.x`, but it could *not* result in `λy.λx.y`. The latter has a different meaning from the original.

Second, alpha-conversion is not possible if it would result in a variable getting captured by a different abstraction. For example, if we replace `x` with `y` in `λx.λy.x`, we get `λy.λy.y`, which is not at all the same.

In programming languages with static scope, alpha-conversion can be used to make name resolution simpler by ensuring that no variable name masks a name in a containing scope.

Substitution

Substitution, written $E[V := E']$, is the process of replacing all free occurrences of the variable V by expression E'. Substitution on terms of the λ-calculus is defined by recursion on the structure of terms, as follows.

$$
\begin{aligned}
x[x := N] &\equiv N \\
y[x := N] &\equiv y, \text{ if } x \neq y \\
(M_1\ M_2)[x := N] &\equiv (M_1[x := N])\ (M_2[x := N]) \\
(\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]), \text{ if } x \neq y \text{ and } y \notin FV(N)
\end{aligned}
$$

To substitute into a lambda abstraction, it is sometimes necessary to α-convert the expression. For example, it is not correct for $(\lambda x.y)[y := x]$ to result in $(\lambda x.x)$, because the substituted x was supposed to be free but ended up being bound. The correct substitution in this case is $(\lambda z.x)$, up-to α-equivalence. Notice that substitution is defined uniquely up-to α-equivalence.

β-reduction

Beta-reduction captures the idea of function application. Beta-reduction is defined in terms of substitution: the beta-reduction of $((\lambda V.E)\ E')$ is $E[V := E']$.

For example, assuming some encoding of $2,\ 7,\ *$, we have the following β-reductions: $((\lambda n.n*2)\ 7) \rightarrow 7*2$.

η-conversion

Eta-conversion expresses the idea of extensionality, which in this context is that two functions are the same if and only if they give the same result for all arguments. Eta-conversion converts between $\lambda x.f\ x$ and f whenever x does not appear free in f.

This conversion is not always appropriate when lambda expressions are interpreted as programs. Evaluation of $\lambda x.f\ x$ can terminate even when evaluation of f does not.

Normal forms and confluence

For the untyped lambda calculus, β-reduction as a rewriting rule is neither strongly normalising nor weakly normalising.

However, it can be shown that β-reduction is confluent. (Of course, we are working up to α-conversion, i.e. we consider two normal forms to be equal if it is possible to α-convert one into the other.)

Therefore, both strongly normalising terms and weakly normalising terms have a unique normal form. For strongly normalising terms, any reduction strategy is guaranteed to yield the normal form, whereas for weakly normalising terms, some reduction strategies may fail to find it.

Encoding datatypes

The basic lambda calculus may be used to model booleans, arithmetic, data structures and recursion, as illustrated in the following sub-sections.

Arithmetic in lambda calculus

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church numerals, which can be defined as follows:

```
0 := λfx.x
1 := λfx.f x
2 := λfx.f (f x)
3 := λfx.f (f (f x))
```

and so on. A Church numeral is a higher-order function—it takes a single-argument function `f`, and returns another single-argument function. The Church numeral `n` is a function that takes a function `f` as argument and returns the `n`-th composition of `f`, i.e. the function `f` composed with itself `n` times. This is denoted $f^{(n)}$ and is in fact the `n`-th power of `f` (considered as an operator); $f^{(0)}$ is defined to be the identity function. Such repeated compositions (of a single function `f`) obey the laws of exponents, which is why these numerals can be used for arithmetic. (In Church's original lambda calculus, the formal parameter of a lambda expression was required to occur at least once in the function body, which made the above definition of `0` impossible.)

We can define a successor function, which takes a number `n` and returns `n + 1` by adding an additional application of `f`:

```
SUCC := λnfx.f (n f x)
```

Because the `m`-th composition of `f` composed with the `n`-th composition of `f` gives the `m+n`-th composition of `f`, addition can be defined as follows:

```
PLUS := λmnfx.m f (n f x)
```

`PLUS` can be thought of as a function taking two natural numbers as arguments and returning a natural number; it can be verified that

`PLUS 2 3` and
`5`

are equivalent lambda expressions. Since adding *m* to a number *n* can be accomplished by adding 1 *m* times, an equivalent definition is:

```
PLUS := λmn.m SUCC n
```

Similarly, multiplication can be defined as

```
MULT := λmnf.m (n f)
```

Alternatively

```
MULT := λmn.m (PLUS n) 0
```

since multiplying *m* and *n* is the same as repeating the add *n* function *m* times and then applying it to zero. Exponentiation has a rather simple rendering in Church numerals, namely

```
POW := λbe.e b
```

The predecessor function defined by `PRED n = n - 1` for a positive integer *n* and `PRED 0 = 0` is considerably more difficult. The formula

```
PRED := λnfx.n (λgh.h (g f)) (λu.x) (λu.u)
```

can be validated by showing inductively that if `T` denotes `(λgh.h (g f))`, then $T^{(n)}(\lambda u.x) = (\lambda h.h(f^{(n-1)}(x)))$ for $n > 0$. Two other definitions of `PRED` are given below, one using conditionals and the other using pairs. With the predecessor function, subtraction is straightforward. Defining

```
SUB := λmn.n PRED m,
```

`SUB m n` yields `m - n` when `m > n` and `0` otherwise.

Logic and predicates

By convention, the following two definitions (known as Church booleans) are used for the boolean values `TRUE` and `FALSE`:

```
TRUE := λxy.x
FALSE := λxy.y
```

(Note that `FALSE` is equivalent to the Church numeral zero defined above)

Then, with these two λ-terms, we can define some logic operators (these are just possible formulations; other expressions are equally correct):

```
AND := λpq.p q p
OR := λpq.p p q
NOT := λpab.p b a
```

```
IFTHENELSE := λpab.p a b
```

We are now able to compute some logic functions, for example:

```
AND TRUE FALSE
≡ (λpq.p q p) TRUE FALSE →β TRUE FALSE TRUE
≡ (λxy.x) FALSE TRUE →β FALSE
```

and we see that `AND TRUE FALSE` is equivalent to `FALSE`.

A *predicate* is a function which returns a boolean value. The most fundamental predicate is `ISZERO` which returns `TRUE` if its argument is the Church numeral `0`, and `FALSE` if its argument is any other Church numeral:

```
ISZERO := λn.n (λx.FALSE) TRUE
```

The following predicate tests whether the first argument is less-than-or-equal-to the second:

```
LEQ := λmn.ISZERO (SUB m n),
```

and since *m = n* iff `LEQ m n` and `LEQ n m`, it is straightforward to build a predicate for numerical equality.

The availability of predicates and the above definition of `TRUE` and `FALSE` make it convenient to write "if-then-else" expressions in lambda calculus. For example, the predecessor function can be defined as:

```
PRED := λn.n (λgk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0) 0
```

which can be verified by showing inductively that `n (λgk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0)` is the add *n* - 1 function for *n* > 0.

Pairs

A pair (2-tuple) can be defined in terms of `TRUE` and `FALSE`, by using the Church encoding for pairs. For example, `PAIR` encapsulates the pair (*x*,*y*), `FIRST` returns the first element of the pair, and `SECOND` returns the second.

```
PAIR := λxyf.f x y
FIRST := λp.p TRUE
SECOND := λp.p FALSE
NIL := λx.TRUE
NULL := λp.p (λxy.FALSE)
```

A linked list can be defined as either NIL for the empty list, or the `PAIR` of an element and a smaller list. The predicate `NULL` tests for the value `NIL`. (Alternatively, with `NIL :=`

FALSE, the construct l (λhtz.deal_with_head_h_and_tail_t) (deal_with_nil) obviates the need for an explicit NULL test).

As an example of the use of pairs, the shift-and-increment function that maps (m, n) to (n, n + 1) can be defined as

```
Φ := λx.PAIR (SECOND x) (SUCC (SECOND x))
```

which allows us to give perhaps the most transparent version of the predecessor function:

```
PRED := λn.FIRST (n Φ (PAIR 0 0)).
```

Recursion and fixed points

Recursion is the definition of a function using the function itself; on the face of it, lambda calculus does not allow this. However, this impression is misleading. Consider for instance the factorial function f(n) recursively defined by

```
f(n) = 1, if n = 0; else n × f(n - 1).
```

In lambda calculus, a function cannot refer directly to itself. However, a function may accept a parameter which is *assumed* to be itself. As an invariant, this argument is typically the first. Binding it to the function yields a new function which may recurse. To achieve recursion, the self-referencing argument (called r here) must always be passed to itself within the function body.

```
g := λr. λn.(1, if n = 0; else n × (r r (n-1)))
f := g g
```

This solves the specific problem of the factorial function, but a generic solution is also possible. Given a lambda term representing the body of a recursive function or loop, taking itself as the first argument, the *fixed-point operator* will return the desired recursive function or loop. The function does not need to be explicitly passed to itself at any point. In fact there are many possible definitions for this operator, generally known as fixed point combinators. The simplest is defined as such:

```
Y = λg.(λx.g (x x)) (λx.g (x x))
```

In the lambda calculus, Y g is a fixed-point of g, as it expands to:

```
Y g
λh.((λx.h (x x)) (λx.h (x x))) g
(λx.g (x x)) (λx.g (x x))
g (λx.g (x x)) (λx.g (x x))
g (Y g).
```

Now, to complete our recursive call to the factorial function, we would simply call `g (Y g) n`, where *n* is the number we are calculating the factorial of. Given *n* = 4, for example, this gives:

```
g (Y g) 4
(λfn.(1, if n = 0; and n·(f(n-1)), if n>0)) (Y g) 4
(λn.(1, if n = 0; and n·((Y g) (n-1)), if n>0)) 4
1, if 4 = 0; and 4·(g(Y g) (4-1)), if 4>0
4·(g(Y g) 3)
4·(λn.(1, if n = 0; and n·((Y g) (n-1)), if n>0) 3)
4·(1, if 3 = 0; and 3·(g(Y g) (3-1)), if 3>0)
4·(3·(g(Y g) 2))
4·(3·(λn.(1, if n = 0; and n·((Y g) (n-1)), if n>0) 2))
4·(3·(1, if 2 = 0; and 2·(g(Y g) (2-1)), if 2>0))
4·(3·(2·(g(Y g) 1)))
4·(3·(2·(λn.(1, if n = 0; and n·((Y g) (n-1)), if n>0) 1)))
4·(3·(2·(1, if 1 = 0; and 1·((Y g) (1-1)), if 1>0)))
4·(3·(2·(1·((Y g) 0))))
4·(3·(2·(1·((λn.(1, if n = 0; and n·((Y g) (n-1)), if n>0) 0))))
4·(3·(2·(1·(1, if 0 = 0; and 0·((Y g) (0-1)), if 0>0))))
4·(3·(2·(1·(1))))
24
```

Every recursively defined function can be seen as a fixed point of some other suitable function, and therefore, using `Y`, every recursively defined function can be expressed as a lambda expression. In particular, we can now cleanly define the subtraction, multiplication and comparison predicate of natural numbers recursively.

Standard terms

Certain terms have commonly accepted names:

```
I := λx.x
K := λxy.x
S := λxyz.(x z (y z))
ω := λx.(x x)
Ω := ω ω
```

Typed lambda calculi

Computable functions and lambda calculus

A function *F*: **N** → **N** of natural numbers is a computable function if and only if there exists a lambda expression *f* such that for every pair of *x*, *y* in **N**, *F*(*x*)=*y* if and only if *f* x $=_\beta$ y, where x and y are the Church numerals corresponding to *x* and *y*, respectively and $=_\beta$ meaning equivalence with beta reduction.

Undecidability of equivalence

There is no algorithm which takes as input two lambda expressions and outputs `TRUE` or `FALSE` depending on whether or not the two expressions are equivalent. This was historically the first problem for which undecidability could be proven. As is common for a proof of undecidability, the proof shows that no computable function can decide the equivalence. Church's thesis is then invoked to show that no algorithm can do so.

Church's proof first reduces the problem to determining whether a given lambda expression has a *normal form*. A normal form is an equivalent expression which cannot be reduced any further under the rules imposed by the form. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. Building on earlier work by Kleene and constructing a Gödel numbering for lambda expressions, he constructs a lambda expression *e* which closely follows the proof of Gödel's first incompleteness theorem. If *e* is applied to its own Gödel number, a contradiction results.

Lambda calculus and programming languages

As pointed out by Peter Landin's 1965 paper *A Correspondence between ALGOL 60 and Church's Lambda-notation*, sequential procedural programming languages can be understood in terms of the lambda calculus, which provides the basic mechanisms for procedural abstraction and procedure (subprogram) application.

Lambda calculus reifies "functions" and makes them first-class objects, which raises implementation complexity when implementing lambda calculus. A particular challenge is related to the support of higher-order functions, also known as the Funarg problem. Lambda calculus is usually implemented using a virtual machine approach. The first practical implementation of lambda calculus was provided in 1963 by Peter Landin, and is known as the SECD machine. Since then, several optimized abstract machines for lambda calculus were suggested, such as the G-machine and the categorical abstract machine.

The most prominent counterparts to lambda calculus in programming are functional programming languages, which essentially implement the calculus augmented with some constants and datatypes. Lisp uses a variant of lambda notation for defining functions, but only its purely functional subset ("Pure Lisp") is really equivalent to lambda calculus.

First-class functions

Anonymous functions are sometimes called lambda expressions in programming languages. An example of the 'square' function as a lambda expression in Lisp:

```
(lambda (x) (* x x))
```

or Haskell:

```
\x -> x*x -- where the \ denotes the greek λ
```

The above Lisp example is an expression which evaluates to a first-class function. The symbol `lambda` creates an anonymous function, given a list of parameter names, `(x)` — just a single argument in this case, and an expression which is evaluated as the body of the function, `(* x x)`. The Haskell example is identical.

Functional languages are not the only ones to support functions as first-class objects. Numerous imperative languages, e.g. Pascal, have long supported passing subprograms as arguments to other subprograms. In C and the C-like subset of C++ the equivalent result is obtained by passing *pointers* to the code of functions (subprograms) deemed function pointers. Such mechanisms are limited to subprograms written explicitly in the code, and do not directly support higher-level functions. Some imperative object-oriented languages have notations that represent functions of any order; such mechanisms are available in C++, Smalltalk and more recently in Eiffel ("agents") and C# ("delegates"). As an example, the Eiffel "inline agent" expression

```
agent (x: REAL): REAL do Result := x * x end
```

denotes an object corresponding to the lambda expression λx.x*x (with call by value). It can be treated like any other expression, e.g. assigned to a variable or passed around to routines. If the value of *square* is the above agent expression, then the result of applying *square* to a value a (β-reduction) is expressed as *square*.item ([a]), where the argument is passed as a tuple.

A Python example of this uses the lambda form of functions:

```
func = lambda x: x ** 2
```

This creates a new anonymous function and names it **func** which can be passed to other functions, stored in variables, etc. Python can also treat any other function created with the standard def statement as first-class objects.

The same holds for Smalltalk expression

```
[ :x | x * x ]
```

This is first-class object (block closure), which can be stored in variables, passed as arguments, etc.

A similar C++ example (using the Boost.Lambda library):

```
std::for_each(c.begin(), c.end(), std::cout << _1 * _1 << '\n');
```

Here the standard library function **for_each** iterates over all members of container 'c', and prints the square of each element. The _1 notation is Boost.Lambda's convention (originally derived from Boost.Bind) for representing the first placeholder element (the first argument), represented as **x** elsewhere.

Reduction strategies

Whether a term is normalising or not, and how much work needs to be done in normalising it if it is, depends to a large extent on the reduction strategy used. The distinction between reduction strategies relates to the distinction in functional programming languages between eager evaluation and lazy evaluation.

Full beta reductions

> Any redex can be reduced at any time. This means essentially the lack of any particular reduction strategy—with regard to reducibility, "all bets are off".

Applicative order

> The leftmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms, even when this is not possible.
> Most programming languages (including Lisp, ML and imperative languages like C and Java) are described as "strict", meaning that functions applied to non-normalising arguments are non-normalising. This is done essentially using applicative order, call by value reduction (see below), but usually called "eager evaluation".

Normal order

> The leftmost, outermost redex is always reduced first. That is, whenever possible the arguments are substituted into the body of an abstraction before the arguments are reduced.

Call by name

> As normal order, but no reductions are performed inside abstractions. For example $\lambda x.(\lambda x.x)x$ is in normal form according to this strategy, although it contains the redex $(\lambda x.x)x$.

Call by value

> Only the outermost redexes are reduced: a redex is reduced only when its right hand side has reduced to a value (variable or lambda abstraction).

Call by need

> As normal order, but function applications that would duplicate terms instead name the argument, which is then reduced only "when it is needed". Called in practical contexts "lazy evaluation". In implementations this "name" takes the form of a pointer, with the redex represented by a thunk.

Applicative order is not a normalising strategy. The usual counterexample is as follows: define $\Omega = \omega\omega$ where $\omega = \lambda x.xx$. This entire expression contains only one redex, namely the whole expression; its reduct is again Ω. Since this is the only available reduction, Ω has no normal form (under any evaluation strategy). Using applicative order, the expression $KI\Omega = (\lambda xy.x)\ (\lambda x.x)\Omega$ is reduced by first reducing Ω to normal form (since it is the leftmost redex), but since `Ω` has no normal form, applicative order fails to find a normal form for `KIΩ`.

In contrast, normal order is so called because it always finds a normalising reduction if one exists. In the above example, `KIΩ` reduces under normal order to `I`, a normal form. A drawback is that redexes in the arguments may be copied, resulting in duplicated computation (for example, `(λx.xx) ((λx.x)y)` reduces to `((λx.x)y) ((λx.x)y)` using this strategy; now there are two redexes, so full evaluation needs two more steps, but if the argument had been reduced first, there would now be none).

The positive tradeoff of using applicative order is that it does not cause unnecessary computation if all arguments are used, because it never substitutes arguments containing redexes and hence never needs to copy them (which would duplicate work). In the above example, in applicative order `(λx.xx) ((λx.x)y)` reduces first to `(λx.xx)y` and then to the normal order `yy`, taking two steps instead of three.

Most *purely* functional programming languages (notably Miranda and its descendents, including Haskell), and the proof languages of theorem provers, use *lazy evaluation*, which is essentially the same as call by need. This is like normal order reduction, but call by need manages to avoid the duplication of work inherent in normal order reduction using *sharing*. In the example given above, `(λx.xx) ((λx.x)y)` reduces to `((λx.x)y) ((λx.x)y)`, which has two redexes, but in call by need they are represented using the same object rather than copied, so when one is reduced the other is too.

A note about complexity

While the idea of beta reduction seems simple enough, it is not an atomic step, in that it must have a non-trivial cost when estimating computational complexity. To be precise, one must somehow find the location of all of the occurrences of the bound variable *`V`* in the expression *`E`*, implying a time cost, or one must keep track of these locations in some way, implying a space cost. A naïve search for the locations of *`V`* in *`E`* is *O*(*n*) in the length *n* of *`E`*. This has led to the study of systems which use explicit substitution. Sinot's director strings offer a way of tracking the locations of free variables in expressions.

Parallelism and concurrency

The Church-Rosser property of the lambda calculus means that evaluation (β-reduction) can be carried out in *any order*, even in parallel. This means that various nondeterministic evaluation strategies are relevant. However, the lambda calculus does not offer any explicit constructs for parallelism. One can add constructs such as Futures to the lambda calculus. Other process calculi have been developed for describing communication and concurrency.

Semantics

The fact that lambda calculus terms act as functions on other lambda calculus terms, and even on themselves, led to questions about the semantics of the lambda calculus. Could a sensible meaning be assigned to lambda calculus terms? The natural semantics was to find a set D isomorphic to the function space $D \rightarrow D$, of functions on itself. However, no nontrivial such D can exist, by cardinality constraints because the set of all functions from D into D has greater cardinality than D.

In the 1970s, Dana Scott showed that, if only continuous functions were considered, a set or domain D with the required property could be found, thus providing a model for the lambda calculus.

This work also formed the basis for the denotational semantics of programming languages.

2. Combinatory logic

Combinatory logic is a notation introduced by Moses Schönfinkel and Haskell Curry to eliminate the need for variables in mathematical logic. It has more recently been used in computer science as a theoretical model of computation and also as a basis for the design of functional programming languages. It is based on **combinators**. A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.

Combinatory logic in mathematics

Combinatory logic was originally intended as a 'pre-logic' that would clarify the role of quantified variables in logic, essentially by eliminating them. Another way of eliminating quantified variables is Quine's predicate functor logic. While the expressive power of combinatory logic typically exceeds that of first-order logic, the expressive power of predicate functor logic is identical to that of first order logic (Quine 1960, 1966, 1976).

The original inventor of combinatory logic, Moses Schönfinkel, published nothing on combinatory logic after his original 1924 paper, and largely ceased to publish after Joseph Stalin consolidated his power in 1929. Curry rediscovered the combinators while working as an instructor at the Princeton University in late 1927. In the latter 1930s, Alonzo Church and his students at Princeton invented a rival formalism for functional abstraction, the lambda calculus, which proved more popular than combinatory logic. The upshot of these historical contingencies was that until theoretical computer science began taking an interest in combinatory logic in the 1960s and 70s, nearly all work on the subject was by Haskell Curry and his students, or by Robert Feys in Belgium. Curry and Feys (1958), and Curry *et al.* (1972) survey the early history of combinatory logic.

Combinatory logic in computing

In computer science, combinatory logic is used as a simplified model of computation, used in computability theory and proof theory. Despite its simplicity, combinatory logic captures many essential features of computation.

Combinatory logic can be viewed as a variant of the lambda calculus, in which lambda expressions (representing functional abstraction) are replaced by a limited set of *combinators*, primitive functions from which free variables are absent. It is easy to transform lambda expressions into combinator expressions, and combinator reduction is much simpler than lambda reduction. Hence combinatory logic has been used to model some non-strict functional programming languages and hardware. The purest form of this view is the programming language Unlambda, whose sole primitives are the S and K combinators augmented with character input/output. Although not a practical programming language, Unlambda is of some theoretical interest.

Combinatory logic can be given a variety of interpretations. Many early papers by Curry showed how to translate axiom sets for conventional logic into combinatory logic equations (Hindley and Meredith 1990). Dana Scott in the 1960s and 70s showed how to marry model theory and combinatory logic.

Summary of the lambda calculus

The lambda calculus is concerned with objects called *lambda-terms*, which are strings of symbols of one of the following forms:

- *v*
- *λv.E1*
- (*E1 E2*)

where *v* is a variable name drawn from a predefined infinite set of variable names, and *E1* and *E2* are lambda-terms.

Terms of the form *λv.E1* are called *abstractions*. The variable *v* is called the formal parameter of the abstraction, and *E1* is the *body* of the abstraction. The term *λv.E1* represents the function which, applied to an argument, binds the formal parameter *v* to the argument and then computes the resulting value of *E1*---that is, it returns *E1*, with every occurrence of *v* replaced by the argument.

Terms of the form *(E1 E2)* are called *applications*. Applications model function invocation or execution: the function represented by *E1* is to be invoked, with *E2* as its argument, and the result is computed. If *E1* (sometimes called the *applicand*) is an abstraction, the term may be *reduced*: *E2*, the argument, may be substituted into the body of $E1$ in place of the formal parameter of $E1$, and the result is a new lambda term which is *equivalent* to the old one. If a lambda term contains no subterms of the form $(\lambda v.E1\ E2)$ then it cannot be reduced, and is said to be in normal form.

The expression $E[v := a]$ represents the result of taking the term E and replacing all free occurrences of v with a. Thus we write

$$(\lambda v.E\ a) \Rightarrow E[v := a]$$

By convention, we take $(a\ b\ c\ d\ ...\ z)$ as short for $(...(((a\ b)\ c)\ d)\ ...\ z)$. (i.e., application is left associative.)

The motivation for this definition of reduction is that it captures the essential behavior of all mathematical functions. For example, consider the function that computes the square of a number. We might write

The square of x is x*x

(Using "*" to indicate multiplication.) x here is the formal parameter of the function. To evaluate the square for a particular argument, say 3, we insert it into the definition in place of the formal parameter:

The square of 3 is 3*3

To evaluate the resulting expression 3*3, we would have to resort to our knowledge of multiplication and the number 3. Since any computation is simply a composition of the evaluation of suitable functions on suitable primitive arguments, this simple substitution principle suffices to capture the essential mechanism of computation. Moreover, in the lambda calculus, notions such as '3' and '*' can be represented without any need for externally defined primitive operators or constants. It is possible to identify terms in the lambda calculus, which, when suitably interpreted, behave like the number 3 and like the multiplication operator.

The lambda calculus is known to be computationally equivalent in power to many other plausible models for computation (including Turing machines); that is, any calculation that can be accomplished in any of these other models can be expressed in the lambda calculus, and vice versa. According to the Church-Turing thesis, both models can express any possible computation.

It is perhaps surprising that lambda-calculus can represent any conceivable computation using only the simple notions of function abstraction and application based on simple textual substitution of terms for variables. But even more remarkable is that abstraction is not even required. *Combinatory logic* is a model of computation equivalent to the lambda calculus, but without abstraction. The advantage of this is that evaluating expressions in lambda calculus is quite complicated because the semantics of substitution must be specified with great care to avoid variable capture problems. In contrast, evaluating expressions in combinatory logic is much simpler, because there is no notion of substitution.

Combinatory calculi

Since abstraction is the only way to manufacture functions in the lambda calculus, something must replace it in the combinatory calculus. Instead of abstraction, combinatory calculus provides a limited set of primitive functions out of which other functions may be built.

Combinatory terms

A combinatory term has one of the following forms:

- x
- P
- $(E_1\ E_2)$

where x is a variable, P is one of the primitive functions, and $(E_1\ E_2)$ is the application of combinatory terms E_1 and E_2. The primitive functions themselves are *combinators*, or functions that, when seen as lambda terms, contain no free variables. To shorten the notations, a general convention is that $(E_1\ E_2\ E_3\ ...\ E_n)$, or even $E_1\ E_2\ E_3 ...\ E_\mathrm{n}$, denotes the term $(...((E_1\ E_2)\ E_3)...\ E_n)$. This is the same general convention as for multiple application in lambda calculus.

Reduction in combinatory logic

In combinatory logic, each primitive combinator comes with a reduction rule of the form

$$(P\ x_1\ ...\ x_n) = E$$

where E is a term mentioning only variables from the set $x_1\ ...\ x_n$. It is in this way that primitive combinators behave as functions.

Examples of combinators

The simplest example of a combinator is **I**, the identity combinator, defined by

$$(\mathbf{I}\ x) = x$$

for all terms x. Another simple combinator is **K**, which manufactures constant functions: $(\mathbf{K}\ x)$ is the function which, for any argument, returns x, so we say

$$((\mathbf{K}\ x)\ y) = x$$

for all terms x and y. Or, following the convention for multiple application,

$(\mathbf{K}\ x\ y) = x$

A third combinator is **S**, which is a generalized version of application:

$(\mathbf{S}\ x\ y\ z) = (x\ z\ (y\ z))$

S applies *x* to *y* after first substituting *z* into each of them. Or put another way, *x* is applied to *y* inside the environment *z*.

Given **S** and **K**, **I** itself is unnecessary, since it can be built from the other two:

$((\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ x)$
$= (\mathbf{S}\ \mathbf{K}\ \mathbf{K}\ x)$
$= (\mathbf{K}\ x\ (\mathbf{K}\ x))$
$= x$

for any term *x*. Note that although $((\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ x) = (\mathbf{I}\ x)$ for any *x*, $(\mathbf{S}\ \mathbf{K}\ \mathbf{K})$ itself is not equal to **I**. We say the terms are extensionally equal. Extensional equality captures the mathematical notion of the equality of functions: that two functions are *equal* if they always produce the same results for the same arguments. In contrast, the terms themselves, together with the reduction of primitive combinators, capture the notion of *intensional equality* of functions: that two functions are *equal* only if they have identical implementations up to the expansion of primitive combinators when these ones are applied to enough arguments. There are many ways to implement an identity function; (**S K K**) and **I** are among these ways. (**S K S**) is yet another. We will use the word *equivalent* to indicate extensional equality, reserving *equal* for identical combinatorial terms.

A more interesting combinator is the fixed point combinator or **Y** combinator, which can be used to implement recursion.

Completeness of the S-K basis

It is a perhaps astonishing fact that **S** and **K** can be composed to produce combinators that are extensionally equal to *any* lambda term, and therefore, by Church's thesis, to any computable function whatsoever. The proof is to present a transformation, *T*[ ], which converts an arbitrary lambda term into an equivalent combinator.

T[ ] may be defined as follows:

1. $T[x] \Rightarrow x$
2. $T[(E_1\ E_2)] \Rightarrow (T[E_1]\ T[E_2])$
3. $T[\lambda x.E] \Rightarrow (\mathbf{K}\ T[E])$ (if *x* does not occur free in *E*)
4. $T[\lambda x.x] \Rightarrow \mathbf{I}$
5. $T[\lambda x.\lambda y.E] \Rightarrow T[\lambda x.T[\lambda y.E]]$ (if *x* occurs free in *E*)
6. $T[\lambda x.(E_1\ E_2)] \Rightarrow (\mathbf{S}\ T[\lambda x.E_1]\ T[\lambda x.E_2])$

This process is also known as *abstraction elimination*.

Conversion of a lambda term to an equivalent combinatorial term

For example, we will convert the lambda term *λx*.*λy*.(*y x*) to a combinator:

T[*λx*.*λy*.(*y x*)]
= *T*[*λx*.*T*[*λy*.(*y x*)]] (by 5)
= *T*[*λx*.(**S** *T*[*λy*.*y*] *T*[*λy*.*x*])] (by 6)
= *T*[*λx*.(**S I** *T*[*λy*.*x*])] (by 4)
= *T*[*λx*.(**S I** (**K** *x*))] (by 3 and 1)
= (**S** *T*[*λx*.(**S I**)] *T*[*λx*.(**K** *x*)]) (by 6)
= (**S** (**K** (**S I**)) *T*[*λx*.(**K** *x*)]) (by 3)
= (**S** (**K** (**S I**)) (**S** *T*[*λx*.**K**] *T*[*λx*.*x*])) (by 6)
= (**S** (**K** (**S I**)) (**S** (**K K**) *T*[*λx*.*x*])) (by 3)
= (**S** (**K** (**S I**)) (**S** (**K K**) **I**)) (by 4)

If we apply this combinator to any two terms *x* and *y*, it reduces as follows:

(**S** (**K** (**S I**)) (**S** (**K K**) **I**) x y)
= (**K** (**S I**) x (**S** (**K K**) **I** x) y)
= (**S I** (**S** (**K K**) **I** x) y)
= (**I** y (**S** (**K K**) **I** x y))
= (y (**S** (**K K**) **I** x y))
= (y (**K K** x (**I** x) y))
= (y (**K** (**I** x) y))
= (y (**I** x))
= (y x)

The combinatory representation, (**S** (**K** (**S I**)) (**S** (**K K**) **I**)) is much longer than the representation as a lambda term, *λx*.*λy*.(y x). This is typical. In general, the *T*[ ] construction may expand a lambda term of length *n* to a combinatorial term of length $\Theta(3^n)$.

Explanation of the *T*[ ] transformation

The *T*[ ] transformation is motivated by a desire to eliminate abstraction. Two special cases, rules 3 and 4, are trivial: *λx*.*x* is clearly equivalent to **I**, and *λx*.*E* is clearly equivalent to (**K** *T*[*E*]) if *x* does not appear free in *E*.

The first two rules are also simple: Variables convert to themselves, and applications, which are allowed in combinatory terms, are converted to combinators simply by converting the applicand and the argument to combinators.

It's rules 5 and 6 that are of interest. Rule 5 simply says that to convert a complex abstraction to a combinator, we must first convert its body to a combinator, and then eliminate the abstraction. Rule 6 actually eliminates the abstraction.

$\lambda x.(E_1\ E_2)$ is a function which takes an argument, say a, and substitutes it into the lambda term $(E_1\ E_2)$ in place of x, yielding $(E_1\ E_2)[x : = a]$. But substituting a into $(E_1\ E_2)$ in place of x is just the same as substituting it into both E_1 and E_2, so

$$(E_1\ \ E_2)[x\ :=\ a]\ =\ (E_1[x\ :=\ a]\ \ E_2[x\ :=\ a])$$

$$\begin{aligned}(\lambda x.(E_1\ E_2)\ a) &= ((\lambda x.E_1\ a)\ (\lambda x.E_2\ a)) \\ &= (\mathbf{S}\ \lambda x.E_1\ \lambda x.E_2\ a) \\ &= ((\mathbf{S}\ \lambda x.E_1\ \lambda x.E_2)\ a)\end{aligned}$$

By extensional equality,

$$\lambda x.(E_1\ \ E2)\qquad =\ (\mathbf{S}\ \lambda x.E_1\ \lambda x.E_2)$$

Therefore, to find a combinator equivalent to $\lambda x.(E_1\ E_2)$, it is sufficient to find a combinator equivalent to $(\mathbf{S}\ \lambda x.E_1\ \lambda x.E_2)$, and

$$(\mathbf{S}\ \ T[\lambda x.E_1]\ \ T[\lambda x.E_2])$$

evidently fits the bill. E_1 and E_2 each contain strictly fewer applications than $(E_1\ E_2)$, so the recursion must terminate in a lambda term with no applications at all—either a variable, or a term of the form $\lambda x.E$.

Simplifications of the transformation

η-reduction

The combinators generated by the T[ ] transformation can be made smaller if we take into account the *η-reduction* rule:

$$T[\lambda x.(E\ \ x)]\ =\ T[E]\qquad (\text{if } x \text{ is not free in } E)$$

$\lambda x.(E$ x) is the function which takes an argument, x, and applies the function E to it; this is extensionally equal to the function E itself. It is therefore sufficient to convert E to combinatorial form.

Taking this simplification into account, the example above becomes:

$$\begin{aligned}& T[\lambda x.\lambda y.(y\ \ x)] \\ =\ & \ldots \\ =\ & (\mathbf{S}\ \ (\mathbf{K}\ \ (\mathbf{S}\ \ \mathbf{I}))\qquad T[\lambda x.(\mathbf{K}\ \ x)])\end{aligned}$$

```
    = (S (K (S I))   K)                    (by η-reduction)
```

This combinator is equivalent to the earlier, longer one:

```
      (S (K (S I))   K x y)
    = (K (S I) x (K x) y)
    = (S I (K x) y)
    = (I y (K x y))
    = (y (K x y))
    = (y x)
```

Similarly, the original version of the *T*[ ] transformation transformed the identity function *λf.λx.*(*f x*) into (**S** (**S** (**K S**) (**S** (**K K**) **I**)) (**K I**)). With the η-reduction rule, *λf.λx.*(*f x*) is transformed into **I**.

One-point basis

There are one-point bases from which every combinator can be composed extensionally equal to *any* lambda term. The simplest example of such a basis is {**X**} where:

```
    X ≡ λx.((xS)K)
```

It is not difficult to verify that:

```
    X (X (X X)) =ηβ K and
    X (X (X (X X))) =ηβ S.
```

Since {**K**, **S**} is a basis, it follows that {**X**} is a basis too. The Iota programming language uses **X** as its sole combinator.

Another simple example of a one-point basis is:

```
    X' ≡ λx.(x K S K) with
    (X' X') X' =β K and
    X' (X' X') =β S
```

X' does not need η contraction in order to produce **K** and **S**.

Combinators B, C

In addition to **S** and **K**, Schönfinkel's paper included two combinators which are now called **B** and **C**, with the following reductions:

```
    (C a b c) = (a c b)
    (B a b c) = (a (b c))
```

He also explains how they in turn can be expressed using only **S** and **K**.

These combinators are extremely useful when translating predicate logic or lambda calculus into combinator expressions. They were also used by Curry, and much later by David Turner, whose name has been associated with their computational use. Using them, we can extend the rules for the transformation as follows:

1. $T[x]$ => x
2. $T[(E_1\ E_2)]$ => $(T[E_1]\ T[E_2])$
3. $T[\lambda x.E]$ => $(\mathbf{K}\ T[E])$ (if x is not free in E)
4. $T[\lambda x.x]$ => $\mathbf{I}$
5. $T[\lambda x.\lambda y.E]$ => $T[\lambda x.T[\lambda y.E]]$ (if x is free in E)
6. $T[\lambda x.(E_1\ E_2)]$ => $(\mathbf{S}\ T[\lambda x.E_1]\ T[\lambda x.E_2])$ (if x is free in both E_1 and E_2)
7. $T[\lambda x.(E_1\ E_2)]$ => $(\mathbf{C}\ T[\lambda x.E_1]\ T[E_2])$ (if x is free in E_1 but not E_2)
8. $T[\lambda x.(E_1\ E_2)]$ => $(\mathbf{B}\ T[E_1]\ T[\lambda x.E_2])$ (if x is free in E_2 but not E_1)

Using **B** and **C** combinators, the transformation of $\lambda x.\lambda y.(y\ x)$ looks like this:

```
  T[λx.λy.(y x)]
= T[λx.T[λy.(y x)]]
= T[λx.(C T[λy.y] x)]      (by rule 7)
= T[λx.(C I x)]
= (C I)                    (η-reduction)
= C*(traditional canonical notation : X* = X I)
= I'(traditional canonical notation: X' = C X)
```

And indeed, $(\mathbf{C}\ \mathbf{I}\ x\ y)$ does reduce to $(y\ x)$:

```
  (C I x y)
= (I y x)
= (y x)
```

The motivation here is that **B** and **C** are limited versions of **S**. Whereas **S** takes a value and substitutes it into both the applicand and its argument before performing the application, **C** performs the substitution only in the applicand, and **B** only in the argument.

The modern names for the combinators come from Haskell Curry's doctoral thesis of 1930. In Schönfinkel's original paper, what we now call **S**, **K**, **I**, **B** and **C** were called **S**, **C**, **I**, **Z**, and **T** respectively.

The reduction in combinator size that results from the new transformation rules can also be achieved without introducing **B** and **C**, as demonstrated in Section 3.2 of .

$\mathbf{CL}_\mathbf{K}$ versus $\mathbf{CL}_\mathbf{I}$ calculus

A distinction must be made between the $\mathbf{CL}_K$ as described here and the $\mathbf{CL}_I$ calculus. The distinction corresponds to that between the λ_K and the λ_I calculus. Unlike the λ_K calculus, the λ_I calculus restricts abstractions to:

$\lambda x.E$ where x has at least one free occurrence in E.

As a consequence, combinator **K** is not present in the λ_I calculus nor in the $\mathbf{CL}_I$ calculus. The constants of $\mathbf{CL}_I$ are: **I**, **B**, **C** and **S**, which form a basis from which all $\mathbf{CL}_I$ terms can be composed (modulo equality). Every λ_I term can be converted into an equal $\mathbf{CL}_I$ combinator according to rules similar to those presented above for the conversion of λ_K terms into $\mathbf{CL}_K$ combinators.

Reverse conversion

The conversion *L*[ ] from combinatorial terms to lambda terms is trivial:

```
L[I]        = λx.x
L[K]        = λx.λy.x
L[C]        = λx.λy.λz.(x z y)
L[B]        = λx.λy.λz.(x (y z))
L[S]        = λx.λy.λz.(x z (y z))
L[(E1 E2)] = (L[E1] L[E2])
```

Note, however, that this transformation is not the inverse transformation of any of the versions of *T*[ ] that we have seen.

Undecidability of combinatorial calculus

A *normal* form is any combinatory term in which the primitive combinators that occur, if any, are not applied to enough arguments to be simplified. It is undecidable whether a general combinatory term has a normal form; whether two combinatory terms are equivalent, etc. This is equivalent to the undecidability of the corresponding problems for lambda terms. However, a direct proof is as follows:

First, observe that the term

```
Ω = (S I I (S I I))
```

has no normal form, because it reduces to itself after three steps, as follows:

```
  (S I I (S I I))
= (I (S I I) (I (S I I)))
= (S I I (I (S I I)))
= (S I I (S I I))
```

and clearly no other reduction order can make the expression shorter.

Now, suppose **N** were a combinator for detecting normal forms, such that

```
(N x) => T, if x has a normal form
         F, otherwise.
```

(Where **T** and **F** represent the conventional Church encodings of true and false, *λx.λy.x* and *λx.λy.y*, transformed into combinatory logic. The combinatory versions have **T** = **K** and **F** = (**K I**).)

Now let

```
Z = (C (C (B N (S I I)) Ω) I)
```

now consider the term (**S I I** *Z*). Does (**S I I** *Z*) have a normal form? It does if and only if the following do also:

```
  (S I I Z)
= (I Z (I Z))
= (Z (I Z))
= (Z Z)
= (C (C (B N (S I I)) Ω) I Z)          (definition of Z)
= (C (B N (S I I)) Ω Z I)
= (B N (S I I) Z Ω I)
= (N (S I I Z) Ω I)
```

Now we need to apply **N** to (**S I I** *Z*). Either (**S I I** *Z*) has a normal form, or it does not. If it *does* have a normal form, then the foregoing reduces as follows:

```
  (N (S I I Z) Ω I)
= (K Ω I)                              (definition of N)
= Ω
```

but **Ω** does *not* have a normal form, so we have a contradiction. But if (**S I I** *Z*) does *not* have a normal form, the foregoing reduces as follows:

```
  (N (S I I Z) Ω I)
= (K I Ω I)                            (definition of N)
= (I I)
  I
```

which means that the normal form of (**S I I** *Z*) is simply **I**, another contradiction. Therefore, the hypothetical normal-form combinator **N** cannot exist.

The combinatory logic analogue of Rice's theorem says that there is no complete nontrivial predicate. A *predicate* is a combinator that, when applied, returns either **T** or **F**. A predicate *N* is *nontrivial* if there are two arguments *A* and *B* such that *NA*=**T** and *NB*=**F**. A combinator *N* is *complete* if and only if *NM* has a normal form for every argument *M*. The analogue of Rice's theorem then says that every complete predicate is trivial. The proof of this theorem is rather simple.

Proof: By reductio ad absurdum. Suppose there is a complete non trivial predicate, say *N*.
Because *N* is supposed to be non trivial there are combinators *A* and *B* such that

(*N A*) = **T** and
(*N B*) = **F**.

Define NEGATION ≡ $\lambda x.$(if (*N x*) then *B* else *A*) ≡ $\lambda x.$((*N x*) *B A*)
Define ABSURDUM ≡ (**Y** NEGATION)

Fixed point theorem gives: ABSURDUM = (NEGATION ABSURDUM), for ABSURDUM ≡ (**Y** NEGATION) = (NEGATION (**Y** NEGATION)) ≡ (NEGATION ABSURDUM).

Because *N* is supposed to be complete either:

1. (*N* ABSURDUM) = **F** or
2. (*N* ABSURDUM) = **T**

Case 1: **F** = (*N* ABSURDUM) = *N* (NEGATION ABSURDUM) = (*N A*) = **T**, a contradiction.
Case 2: **T** = (*N* ABSURDUM) = *N* (NEGATION ABSURDUM) = (*N B*) = **F**, again a contradiction.

Hence (*N* ABSURDUM) is neither **T** nor **F**, which contradicts the presupposition that *N* would be a complete non trivial predicate. **QED**.

From this undecidability theorem it immediately follows that there is no complete predicate that can discriminate between terms that have a normal form and terms that do not have a normal form. It also follows that there is **no** complete predicate, say EQUAL, such that:
(EQUAL *A B*) = **T** if *A* = *B* and
(EQUAL *A B*) = **F** if $A \neq B$.
If EQUAL would exist, then for all *A*, $\lambda x.$(EQUAL *x A*) would have to be a complete non trivial predicate.

3. μ-recursive function

In mathematical logic and computer science, the **μ-recursive functions** are a class of partial functions from natural numbers to natural numbers which are "computable" in an intuitive sense. In fact, in computability theory it is shown that the μ-recursive functions are precisely the functions that can be computed by Turing machines. The μ-recursive functions are closely related to primitive recursive functions, and their inductive definition (below) builds upon that of the primitive recursive functions. However, not every μ-recursive function is a primitive recursive function — the most famous example is the Ackermann function.

Other equivalent classes of functions are the λ-recursive functions and the functions that can be computed by Markov algorithms.

The set of all recursive functions is known as R in computational complexity theory.

Definition

The **μ-recursive functions** (or **partial μ-recursive functions**) are partial functions that take finite tuples of natural numbers and return a single natural number. They are the smallest class of partial functions that includes the initial functions and is closed under composition, primitive recursion, and the μ operator.

The smallest class of functions including the initial functions and closed under composition and primitive recursion (i.e. without minimisation) is the class of primitive recursive functions. While all primitive recursive functions are total, this is not true of partial recursive functions; for example, the minimisation of the successor function is undefined. The set of total recursive functions is a subset of the partial recursive functions and is a superset of the primitive recursive functions; functions like the Ackermann function can be proven to be total recursive, and not primitive.

The first three functions are called the "initial" or "basic" functions: (In the following the subscripting is per Kleene (1952) p. 219.)

1. **Constant function**: For each natural number $n$and every k:

   $f(x_1,\ldots,x_k) = n$.
   Alternative definitions use compositions of the successor function and use a **zero function**, that always returns zero, in place of the constant function.

2. **Successor function S:**

   $S(x) =_{def} f(x) = x + 1$

3. **Projection function** P_i^k (also called the **Identity function** I_i^k): For all natural numbers i, ksuch that $1 \le i \le k$:

   $P_i^k =_{def} f(x_1,\ldots,x_k) = x_i$.

4. **Composition operator** $\circ$(also called the **substitution operator**): Given an m-ary function $h(x_1,\ldots,x_m)$and m k-ary functions $g_1(x_1,\ldots,x_k),\ldots,g_m(x_1,\ldots,x_k)$:

   $h \circ (g_1,\ldots,g_m) =_{def} f(x_1,\ldots,x_k) = h(g_1(x_1,\ldots,x_k),\ldots,g_m(x_1,\ldots,x_k))$.

5. **Primitive recursion operator** ρ: Given the k-ary function $g(x_1,\ldots,x_k)$and k+2 -ary function $h(y,z,x_1,\ldots,x_k)$:

$$\begin{aligned}\rho(g,h) &=_{def} f(y,x_1,\ldots,x_k) \quad \text{where} \\ f(0,x_1,\ldots,x_k) &= g(x_1,\ldots,x_k) \\ f(y+1,x_1,\ldots,x_k) &= h(y,f(y,x_1,\ldots,x_k),x_1,\ldots,x_k).\end{aligned}$$

6. **Minimisation operator** μ: Given a (k+1)-ary partial function $f(y,x_1,\ldots,x_k)$:

$$\begin{aligned}\mu(f)(x_1,\ldots,x_k) = z \iff_{def} &\exists y_0,\ldots,y_z \quad \text{such that} \\ &y_i = f(i,x_1,\ldots,x_k) \quad \text{for } i=0,\ldots,z \\ &y_i > 0 \quad \text{for } i=0,\ldots,z-1 \\ &y_z = 0\end{aligned}$$

   Intuitively, minimisation finds the smallest argument that causes the function to return zero, providing that the function is defined for all smaller arguments.

The **strong equality** operator $\simeq$ can be used to compare partial μ-recursive functions. This is defined for all partial functions *f* and *g* so that

$$f(x_1,\ldots,x_k) \simeq g(x_1,\ldots,x_l)$$

holds if and only if for any choice of arguments either both functions are defined and their values are equal or both functions are undefined.

Equivalence with other models of computability

In the equivalence of models of computability, a parallel is drawn between Turing machines which do not terminate for certain inputs and an undefined result for that input in the corresponding partial recursive function. The unbounded search operator is not definable by the rules of primitive recursion as those do not provide a mechanism for "infinite loops" (undefined values).

Normal form theorem

A **normal form theorem** due to Kleene says that for each *k* there are primitive recursive functions $U(y)$ and $T(y,e,x_1,\ldots,x_k)$ such that for any μ-recursive function $f(x_1,\ldots,x_k)$ with *k* free variables there is an *e* such that

$$f(x_1,\ldots,x_k) \simeq U(\mu y T(y,e,x_1,\ldots,x_k)).$$

The number *e* is called an **index** or **Gödel number** for the function *f*. A consequence of this result is that any μ-recursive function can be defined using a single instance of the μ operator applied to a (total) primitive recursive function.

Minsky (1967) observes (as does Boolos-Burgess-Jeffrey (2002) pp. 94–95) that the U defined above is in essence the μ-recursive equivalent of the universal Turing machine:

> To construct U is to write down the definition of a general-recursive function U(n, x) that correctly interprets the number n and computes the appropriate function of x. to construct U directly would involve essentially the same amount of effort, *and essentially the same ideas*, as we have invested in constructing the universal Turing machine. (italics in original, Minsky (1967) p. 189)

Symbolism

A number of different symbolisms are used in the literature. An advantage to using the symbolism is a derivation of a function by "nesting" of the operators one inside the other is easier to write in a compact form. In the following we will abbreviate the string of parameters $x_1, ..., x_n$ as **x**:

- **Constant function**: Kleene uses " $C_q^n(\mathbf{x}) = q$ " and Boolos-Burgess-Jeffry (2002) (B-B-J) use the abbreviation " $\text{const}_n(\mathbf{x}) = n$ ":

  e.g. C_{13}^7 (r, s, t, u, v, w, x) = 13
  e.g. const_{13} (r, s, t, u, v, w, x) = 13

- **Successor function**: Kleene uses x' and S for "Successor". As "successor" is considered to be primitive, most texts use the apostrophe as follows:

  $S(a) = a +1 =_{def} a'$, where $1 =_{def} 0'$, $2 =_{def} 0 ' '$, etc.

- **Identity function**: Kleene (1952) uses " U_i^n " to indicate the identity function over the variables x_i; B-B-J use the identity function id_i^n over the variables x_1 to x_n:

  $U_i^n(\mathbf{x}) = id_i^n(\mathbf{x}) = x_i$
  e.g. $U_3^7 = id_3^7$ (r, s, t, u, v, w, x) = t

- **Composition (Substitution) operator**: Kleene uses a bold-face $\mathbf{S}_n^m$ (not to be confused with his S for "successor" **!**). The superscript "m" refers to the mth of function "f_m", whereas the subscript "n" refers to the nth variable "x_n":

  If we are given $h(\mathbf{x})= g(f_1(\mathbf{x}), ... , f_m(\mathbf{x}))$
  $h(\mathbf{x}) = \mathbf{S}_m^n(g, f_1, ... , f_m)$
  In a similar manner, but without the sub- and superscripts, B-B-J write:
  $h(x')= Cn[g, f_1 ,..., f_m](\mathbf{x})$

- **Primitive Recursion**: Kleene uses the symbol " $\mathbf{R}^n$(base step, induction step) " where n indicates the number of variables, B-B-J use " Pr(base step, induction step)(**x**)". Given:

- base step: $h(0, \mathbf{x})= f(\mathbf{x})$, and
- induction step: $h(y+1, \mathbf{x}) = g(y, h(x,y),\mathbf{x})$

Example: primitive recursion definition of a + b:

- base step: $f(0, a) = a = U_1^1(a)$
- induction step: $f(b' , a) = (f (b, a))' = g(b, f(b, a), a) = g(b, c, a) = c' = S(U_2^3(b, c, a)$

$R^2 \{ U_1^1(a), S [(U_2^3(b, c, a)] \}$
$Pr\{ U_1^1(a), S[(U_2^3(b, c, a)] \}$

Example: Kleene gives an example of how to perform the recursive derivation of f(b, a) = b + a (notice reversal of variables a and b). He starting with 3 initial functions

1. $S(a) = a'$
2. $U_1^1(a) = a$
3. $U_2^3(b, c, a) = c$
4. $g(b, c, a) = S(U_2^3(b, c, a)) = c'$
5. base step: $h(0, a) = U_1^1(a)$

induction step: $h(b', a) = g(b, h(b, a), a)$

He arrives at:

$a+b = \mathbf{R}^2[U_1^1, \mathbf{S}_1^3(S, U_2^3)]$

4. Markov algorithm

A **Markov algorithm** is a string rewriting system that uses grammar-like rules to operate on strings of symbols. Markov algorithms have been shown to be Turing-complete, which means that they are suitable as a general model of computation and can represent any mathematical expression from its simple notation.

Refal is a programming language based on **Markov algorithm**.

Algorithm

The *Rules* is a sequence of pair of strings, usually presented in the form of *pattern* → *replacement*. Some rules may be terminating.

Given an *input* string:

1. Check the Rules in order from top to bottom to see whether any of the *patterns* can be found in the *input* string.

2. If none is found, the algorithm stops.
3. If one (or more) is found, use **the first** of them to replace the leftmost matching text in the *input* string with its *replacement*.
4. If the applied rule was a terminating one, the algorithm stops.
5. Return to step 1 and carry on.

Example

The following example shows the basic operation of a Markov algorithm.

Rules

1. "A" -> "apple"
2. "B" -> "bag"
3. "S" -> "shop"
4. "T" -> "the"
5. "the shop" -> "my brother"
6. "a never used" -> **.**"terminating rule"

Symbol string

"I bought a B of As from T S."

Execution

If the algorithm is applied to the above example, the Symbol string will change in the following manner.

1. "I bought a B of apples from T S."
2. "I bought a bag of apples from T S."
3. "I bought a bag of apples from T shop."
4. "I bought a bag of apples from the shop."
5. "I bought a bag of apples from my brother."

The algorithm will then terminate.

Another Example

These rules give a more interesting example. They rewrite binary numbers to their unary counterparts. For example: 101 will be rewritten to a string of 5 consecutive bars.

Rules

1. "|0" -> "0||"
2. "1" -> "0|"
3. "0" -> ""

Symbol string

"101"

Execution

If the algorithm is applied to the above example, it will terminate after the following steps.

1. "0|01"
2. "00||1"
3. "00||0|"
4. "00|0|||"
5. "000|||||"
6. "00|||||"
7. "0|||||"
8. "|||||"

5. Register machine

In mathematical logic and theoretical computer science a **register machine** is a generic class of abstract machines used in a manner similar to a Turing machine. All the models are Turing equivalent.

Overview

The register machine gets its name from its one or more "registers" – in place of a Turing machine's tape and head (or tapes and heads) the model uses **multiple, uniquely-addressed registers**, each of which holds a single positive integer.

There are at least 4 sub-classes found in the literature, here listed from most primitive to the most like a computer:

- counter machine – the most primitive and reduced model. Lacks indirect addressing. Instructions are in the finite state machine in the manner of the Harvard architecture.
- Pointer machine – a blend of counter machine and RAM models. Less common and more abstract than either model. Instructions are in the finite state machine in the manner of the Harvard architecture.
- Random access machine (RAM) – a counter machine with indirect addressing and, usually, an augmented instruction set. Instructions are in the finite state machine in the manner of the Harvard architecture.

- Random access stored program machine model (RASP) – a RAM with instructions in its registers analogous to the Universal Turing machine; thus it is an example of the von Neumann architecture. But unlike a computer the model is *idealized* with effectively-infinite registers (and if used, effectively-infinite special registers such as an accumulator). Unlike a computer or even a RISC, the instruction set is much reduced in the number of instructions.

Any properly-defined register machine model is Turing equivalent. Computational speed is very dependent on the model specifics.

In practical computer science, a similar concept known as a virtual machine is sometimes used to minimise dependencies on underlying machine architectures. Such machines are also used for teaching. The term "register machine" is sometimes used to refer to a virtual machine in textbooks.

Formal definition

> *No standard terminology exists; each author is responsible for defining in prose the meanings of their mnemonics or symbols. Many authors use a "register-transfer"-like symbolism to explain the actions of their models, but again they are responsible for defining its syntax.*

A register machine consists of:

1. **An unbounded number of labeled, discrete, unbounded registers unbounded in extent (capacity)**: a finite (or infinite in some models) set of registers $r_0 \ldots r_n$each considered to be of infinite extent and each of which holds a single non-negative integer (0, 1, 2, ...). The registers may do their own arithmetic, or there may be one or more special registers that do the arithmetic e.g. an "accumulator" and/or "address register".
2. **Tally counters or marks**: discrete, indistinguishable objects or marks of only one sort suitable for the model. In the most-reduced counter machine model, per each arithmetic operation only one object/mark is either added to or removed from its location/tape. In some counter machine models (e.g. Melzak (1961), Minsky (1961)) and most RAM and RASP models more than one object/mark can be added or removed in one operation with "addition" and usually "subtraction"; sometimes with "multiplication" and/or "division". Some models have control operations such as "copy" (variously: "move", "load", "store") that move "clumps" of objects/marks from register to register in one action.
3. **A (very) limited set of instructions**: the instructions tend to divide into two classes: arithmetic and control. The instructions are drawn from the two classes to form "instruction-sets", such that an instruction set must allow the model to be Turing equivalent (it must be able to compute any partial recursive function).
    1. **Arithmetic**: arithmetic instructions may operate on all registers or on just a special register (e.g. accumulator). They are *usually* chosen from the following sets (but exceptions abound):

        - Counter machine: { Increment (r), Decrement (r), Clear-to-zero (r) }
        - Reduced RAM, RASP: { Increment (r), Decrement (r), Clear-to-zero (r), Load-immediate-constant k, Add (r_1,r_2), proper-Subtract (r_1,r_2), Increment accumulator, Decrement accumulator, Clear accumulator, Add to accumulator contents of register r, proper-Subtract from accumulator contents of register r, }
        - Augmented RAM, RASP: All of the reduced instructions plus: { Multiply, Divide, various Boolean bit-wise (left-shift, bit test, etc.)}
    2. **Control**:
        - Counter machine models: optional { Copy (r_1,r_2) }
        - RAM and RASP models: most have { Copy (r_1,r_2) }, or { Load Accumulator from r, Store accumulator into r, Load Accumulator with immediate constant }
        - All models: at least one *conditional "jump"* (branch, goto) following test of a register e.g. { Jump-if-zero, Jump-if-not-zero (i.e. Jump-if-positive), Jump-if-equal, Jump-if-not equal }
        - All models optional: { unconditional program jump (goto) }
    3. **Register-addressing method**:
        - Counter machine: no indirect addressing, immediate operands possible in highly atomized models
        - RAM and RASP: indirect addressing available, immediate operands typical
    4. **Input-output**: optional in all models
4. **State register**: A special Instruction Register "IR", finite and separate from the registers above, stores the current instruction to be executed and its address in the TABLE of instructions; this register and its TABLE is located in the finite state machine.
    - The IR is off-limits to all models. In the case of the RAM and RASP, for purposes of determining the "address" of a register, the model can select either (i) in the case of direct addressing—the address specified by the TABLE and temporarily located in the IR or (ii) in the case of indirect addressing—the contents of the register specified by the IR's instruction.
    - The IR is *not* the "program counter" (PC) of the RASP (or conventional computer). The PC is just another register similar to an accumulator, but dedicated to holding the number of the RASP's current register-based instruction. Thus a RASP has *two* "instruction/program" registers -- (i) the IR (finite state machine's Instruction Register), and (ii) a PC (Program Counter) for the program located in the registers. (As well as a register dedicated to "the PC", a RASP may dedicate another register to "the Program-Instruction Register" (going by any number of names such as "PIR, "IR", "PR", etc.)
5. **List of labeled instructions, usually in sequential order**: A finite list of instructions $I_1 \ldots I_m$. In the case of the counter machine, random access machine (RAM) and pointer machine the instruction store is in the "TABLE" of

the finite state machine; thus these models are example of the Harvard architecture. In the case of the RASP the program store is in the registers; thus this is an example of the von Neumann architecture
Usually, like computer programs, the instructions are listed in sequential order; unless a jump is successful the default sequence continues in numerical order. An exception to this is the abacus (Lambek (1961), Minsky (1961)) counter machine models—every instruction has at least one "next" instruction identifier "z", and the conditional branch has two.

- Observe also that the abacus model combines two instructions, JZ then DEC: e.g. { INC (r, z), JZDEC (r, z_{true}, z_{false}) }.

Historical development of the register machine model

Two trends appeared in the early 1950s—the first to characterize the computer as a Turing machine, the second to define computer-like models—models with sequential instruction sequences and conditional jumps—with the power of a Turing machine, i.e. a so-called Turing equivalence. Need for this work was carried out in context of two "hard" problems: the unsolvable word problem posed by Emil Post -- his problem of "tag" -- and the very "hard" problem of Hilbert's problems -- the 10th question around Diophantine equations. Researchers were questing for Turing-equivalent models that were less "logical" in nature and more "arithmetic" (cf Melzak (1961) p. 281, Shepherdson-Sturgis (1963) p. 218).

The first trend—toward characterizing computers—seems to have originated with Hans Hermes (1954) and Heinz Kaphengst (1959), the second trend with Hao Wang (1954, 1957) and, as noted above, furthered along by Z. A.Melzak (1961), Joachim Lambek (1961), Marvin Minsky (1961, 1967), and John Shepherdson and H. E. Sturgis (1963).

The last five names are listed explicitly in that order by Yuri Matiyasevich. He follows up with:

"Register machines [some authors use "register machine" synonymous with "counter-machine"] are particularly suitable for constructing Diophantine equations. Like Turing machines, they have very primitive instructions and, in addition, they deal with numbers"**.**

It appears that Lambek, Melzak, Minsky and Shepherdson and Sturgis independently anticipated the same idea at the same time.

The history begins with Wang's model.

(1954, 1957) Wang's model: Post-Turing machine

Wang's work followed from Emil Post's (1936) paper and led Wang to his definition of his Wang B-machine—a two-symbol Post-Turing machine computation model with only four atomic instructions:

{ LEFT, RIGHT, PRINT, JUMP_if_marked_to_instruction_z }

To these four both Wang (1954, 1957) and then C.Y. Lee (1961) added another other instruction from the Post set { ERASE }, and then a Post's unconditional jump { JUMP_to_ instruction_z } (or to make things easier, the conditional jump JUMP_IF_blank_to_instruction_z, or both. Lee named this a "W-machine" model:

{ LEFT, RIGHT, PRINT, ERASE, JUMP_if_marked, [maybe JUMP or JUMP_IF_blank] }

Wang expressed hope that his model would be "a rapprochement" (p. 63) between the theory of Turing machines and the practical world of the computer.

Wang's work was highly influential. We find him referenced by Minsky (1961) and (1967), Melzak (1961), Shepherdson and Sturgis (1963). Indeed, Shepherdson and Sturgis (1963) remark that:

> "...we have tried to carry a step further the 'rapprochement' between the practical and theoretical aspects of computation suggested by Wang" (p. 218)

Martin Davis eventually evolved this model into the (2-symbol) Post-Turing machine.

Difficulties with the Wang/Post-Turing model:

Except there was a problem: the Wang model (the six instructions of the 7-instruction Post-Turing machine) was still a single-tape Turing-like device, however nice its *sequential program instruction-flow* might be. Both Melzak (1961) and Shepherdson and Sturgis (1963) observed this (in the context of certain proofs and investigations):

"...a Turing machine has a certain opacity... a Turing machine is slow in (hypothetical) operation and, usually, complicated. This makes it rather hard to design it, and even harder to investigate such matters as time or storage optimization or a comparison between efficiency of two algorithms. (Melzak (1961) p. 281)
"...although not difficult ... proofs are complicated and tedious to follow for two reasons: (1) A Turing machine has only head so that one is obliged to break down the computation into very small steps of operations on a single digit. (2) It has only one tape so that one has to go to some trouble to find the under one wishing to work on and keep it separate from other numbers" (Shepherdson and Sturgis (1963) p. 218).

Indeed as examples at Turing machine examples, Post-Turing machine and partial function show, the work can be "complicated".

Minsky, Melzak-Lambek and Shepherdson-Sturgis models "cut the tape" into many

So why not 'cut the tape' so each is infinitely long (to accommodate any size integer) but left-ended, and call these three tapes "Post-Turing (ie. Wang-like) tapes"? The individual heads will move left (for decrement) and right (for increment). In one sense the heads indicate "the tops of the stack" of concatenated marks. Or in Minsky (1961) and Hopcroft and Ullman (1979, p. 171ff) the tape is always blank except for a mark at the left end—at no time does a head ever print or erase.

We just have to be careful to write our instructions so that a test-for-zero and jump occurs *before* we decrement otherwise our machine will "fall off the end" or "bump against the end" -- we will have an instance of a partial function. Before a decrement our machine must always ask the question: "Is the tape/counter empty? If so then I can't decrement, otherwise I can."

Minsky (1961) and Shepherdson-Sturgis (1963) prove that only a few tapes—as few as one—still allow the machine to be Turing equivalent *IF* the data on the tape is represented as a Gödel number (or some other uniquely encodable-decodable number); this number will evolve as the computation proceeds. In the one tape version with Gödel number encoding the counter machine must be able to (i) multiply the Gödel number by a constant (numbers "2" or "3"), and (ii) divide by a constant (numbers "2" or "3") and jump if the remainder is zero. Minsky (1967) shows that the need for this bizarre instruction set can be relaxed to { INC (r), JZDEC (r, z) } and the convenience instructions { CLR (r), J (r) } if two tapes are available. A simple Gödelization is still required, however. A similar result appears in Elgot-Robinson (1964) with respect to their RASP model.

(1961) Melzak's model is different: clumps of pebbles go into and out of holes

Melzak's (1961) model is significantly different. He took his own model, flipped the tapes vertically, called them "holes in the ground" to be filled with "pebble counters". Unlike Minsky's "increment" and "decrement", Melzak allowed for proper subtraction of any count of pebbles and "adds" of any count of pebbles.

He defines indirect addressing for his model (p. 288) and provides two examples of its use (p. 89); his "proof" (p. 290-292) that his model is Turing equivalent is so sketchy that the reader cannot tell whether or not he intended the indirect addressing to be a requirement for the proof.

Legacy of Melzak's model is Lambek's simplification and the reappearance of his mnemonic conventions in Cook and Reckhow 1973.

Lambek (1961) atomizes Melzak's model into the Minsky (1961) model: INC and DEC-with-test

Lambek (1961) took Melzak's ternary model and atomized it down to the two unary instructions—X+, X- if possible else jump—exactly the same two that Minsky (1961) had come up with.

However, like the Minsky (1961) model, the Lambek model does execute its instructions in a default-sequential manner—both X+ and X- carry the identifier of the next instruction, and X- also carries the jump-to instruction of the zero-test is successful.

Elgot-Robinson (1964) and the problem of the RASP without indirect addressing

A RASP or Random access stored program machine begins as a counter machine with its "program of instruction" placed in its "registers". Analogous to, but independent of, the finite state machine's "Instruction Register", at least one of the registers (nicknamed the "program counter" (PC)) and one or more "temporary" registers maintain a record of, and operate on, the current instruction's number. The finite state machine's TABLE of instructions is responsible for (i) fetching the current *program* instruction from the proper register, (ii) parsing the *program* instruction, (ii) fetching operands specified by the *program* instruction, and (iv) executing the *program* instruction.

Except there is a problem: If based on the *counter machine* chassis this computer-like, von Neumann machine will not be Turing equivalent. It cannot compute everything that is computable. Intrinsically the model is bounded by the size of its (very-) *finite* state machine's instructions. The counter machine based RASP can compute any primitive recursive function (e.g. multiplication) but not all mu recursive functions (e.g. the Ackermann function).

Elgot-Robinson investigate the possibility of allowing their RASP model to "self modify" its program instructions. The idea was an old one, proposed by Burks-Goldstine-von Neumann (1946-7), and sometimes called "the computed goto." Melzak (1961) specifically mentions the "computed goto" by name but instead provides his model with indirect addressing.

Computed goto: A RASP *program* of instructions that modifies the "goto address" in a conditional- or unconditional-jump *program* instruction.

But this does not solve the problem (unless one resorts to Gödel numbers). What is necessary is a method to fetch the address of a program instruction that lies (far) "beyond/above" the upper bound of the *finite* state machine's instruction register and TABLE.

Example: A counter machine equipped with only four unbounded registers can e.g. multiply any two numbers (m, n) together to yield p -- and thus be a primitive recursive

function -- no matter how large the numbers m and n; moreover, less than 20 instructions are required to do this! e.g. { 1: CLR (p), 2: JZ (m, done), 3 outer_loop: JZ (n, done), 4: CPY (m, temp), 5: inner_loop: JZ (m, outer_loop), 6: DEC (m), 7: INC (p), 8: J (inner_loop), 9: outer_loop: DEC (n), 10 J (outer_loop), HALT }
However, with only 4 registers, this machine has not nearly big enough to build a RASP that can execute the multiply algorithm as a *program*. No matter how big we build our finite state machine there will always be a *program* (including its parameters) which is larger. So by definition the bounded program machine that does not use unbounded encoding tricks such as Gödel numbers cannot not *universal*.

Minsky (1967) hints at the issue in his investigation of a counter machine (he calls them "program computer models") equipped with the instructions { CLR (r), INC (r), and RPT ("a" times the instructions m to n) }. He doesn't tell us how to fix the problem, but he does observe that:

"... the program computer has to have some way to keep track of how many RPT's remain to be done, and this might exhaust any particular amount of storage allowed in the finite part of the computer. RPT operations require infinite registers of their own, in general, and they must be treated differently from the other kinds of operations we have considered." (p. 214)

But Elgot and Robinson solve the problem: They augment their P_0 RASP with an indexed set of instructions—a somewhat more complicated (but more flexible) form of indirect addressing. Their P'_0 model addresses the registers by adding the contents of the "base" register (specified in the instruction) to the "index" specified explicitly in the instruction (or vice versa, swapping "base" and "index"). Thus the indexing P'_0 instructions have one more parameter than the non-indexing P_0 instructions:

Example: INC (r_{base}, index) ; effective address will be [r_{base}] + index, where the natural number "index" is derived from the finite-state machine instruction itself.

Hartmanis (1971)

By 1971 Hartmanis has simplified the indexing to indirection for use in his RASP model.

Indirect addressing: A pointer-register supplies the finite state machine with the address of the target register required for the instruction. Said another way: The *contents* of the pointer-register is the *address* of the "target" register to be used by the instruction. If the pointer-register is unbounded, the RAM, and a suitable RASP built on its chassis, will be Turing equivalent. The target register can serve either as a source or destination register, as specified by the instruction.

Note that the finite state machine does not have to explicitly specify this target register's address. It just says to the rest of the machine: Get me the contents of the register pointed to by my pointer-register and then do xyz with it. It must specify explicitly by name, via its instruction, this pointer-register (e.g. "N", or "72" or "PC", etc.) but it doesn't have to know what number the pointer-register actually contains (perhaps 279,431).

Cook and Reckhow (1973) describe the RAM

Cook and Reckhow (1973) cite Hartmanis (1971) and simplify his model to what they call a Random access machine (RAM—i.e. a machine with indirection and the Harvard architecture). In a sense we are back to Melzak (1961) but with a much simpler model than Melzak's.

Precedence

Minsky was working at the M.I.T. Lincoln Labs and published his work there; his paper was received for publishing in the *Annals of Mathematics* on August 15, 1960 but not published until November 1961. While receipt occurred a full year before the work of Melzak and Lambek was received and published (received, respectively, May and June 15, 1961 and published side-by-side September 1961). That (i) both were Canadians and published in the Canadian Mathematical Bulletin, (ii) neither would have had reference to Minsky's work because it was not yet published in a peer-reviewed journal, but (iii) Melzak references Wang, and Lambek references Melzak, leads one to hypothesize that their work occurred simultaneously and independently.

Almost exactly the same thing happened to Shepherdson and Sturgis. Their paper was received in December 1961—just a few months after Melzak and Lambek's work was received. Again, they had little (at most 1 month) or no benefit of reviewing the work of Minsky. They were careful to observe in footnotes that papers by Ershov, Kaphengst and Peter had "recently appeared" (p. 219). These were published much earlier but appeared in the German language in German journals so issues of accessibility present themselves.

The final paper of Shepherdson and Sturgis did not appear in a peer-reviewed journal until 1963. And as they fairly and honestly note in their Appendix A, the 'systems' of Kaphengst (1959), Ershov (1958), Peter (1958) are all so similar to what results were obtained later as to be indistinguishable to a set of the following:

produce 0 i.e. 0 --> n
increment a number i.e. n+1 --> n
"i.e. of performing the operations which generate the natural numbers" (p. 246)
copy a number i.e. n --> m
to "change the course of a computation", either comparing two numbers or decrementing until 0

Indeed, Shepherson and Sturgis conclude

"The various minimal systems are very similar"(p. 246)

By order of *publishing* date the work of Kaphengst (1959), Ershov (1958), Peter (1958) were first. Does context matter? An answer would require close examination of the papers. Conclusions and opinions about this will be left to the reader.

Chapter 6

Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism—with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to getting good parallel program performance.

The speed-up of a program as a result of parallelization is observed as Amdahl's law.

Background

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next is executed.

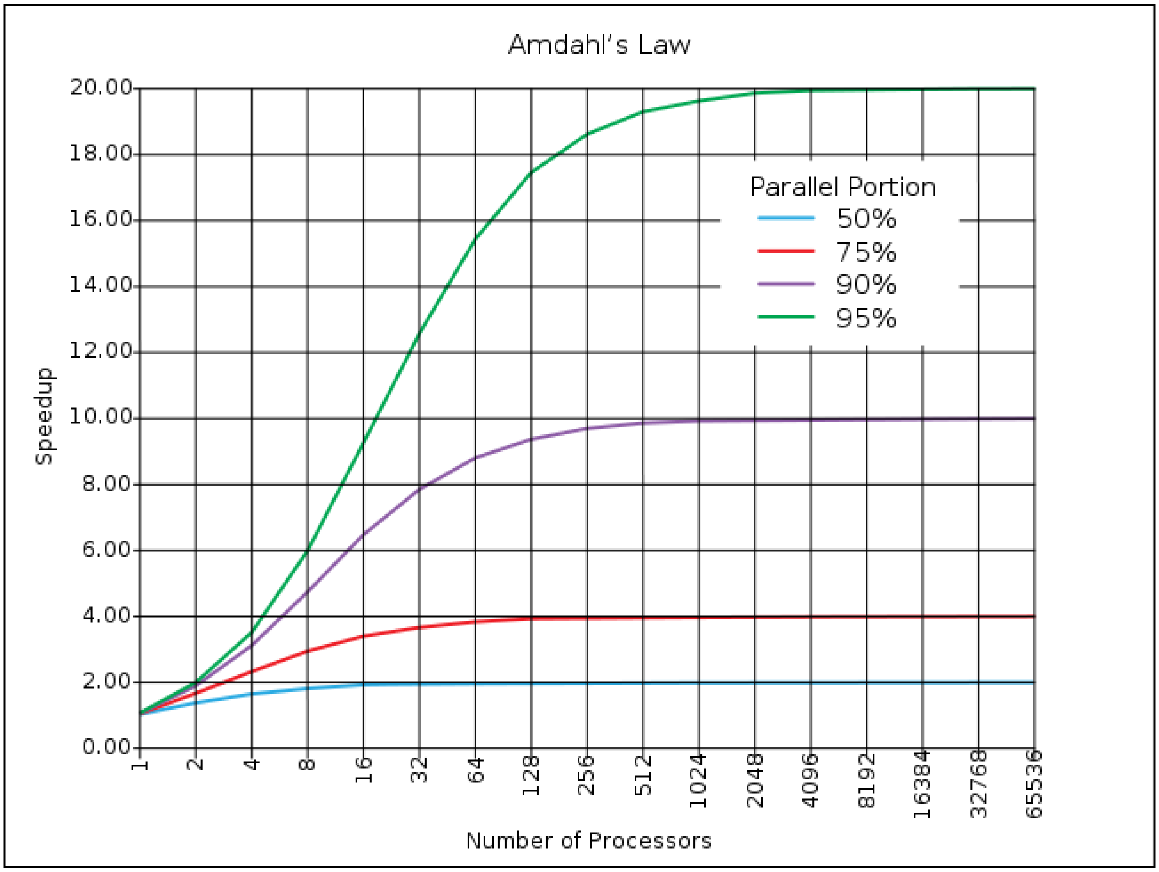
Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all computation-bounded programs.

However, power consumption by a chip is given by the equation $P = C \times V^2 \times F$, where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and F is the processor frequency (cycles per second). Increases in frequency increase the amount of power used in a processor. Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

Moore's Law is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. Despite power consumption issues, and repeated predictions of its end, Moore's law is still in effect. With the end of frequency scaling, these additional transistors (which are no longer used for frequency scaling) can be used to add extra hardware for parallel computing.

Amdahl's law and Gustafson's law



A graphical representation of Amdahl's law. The speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used.

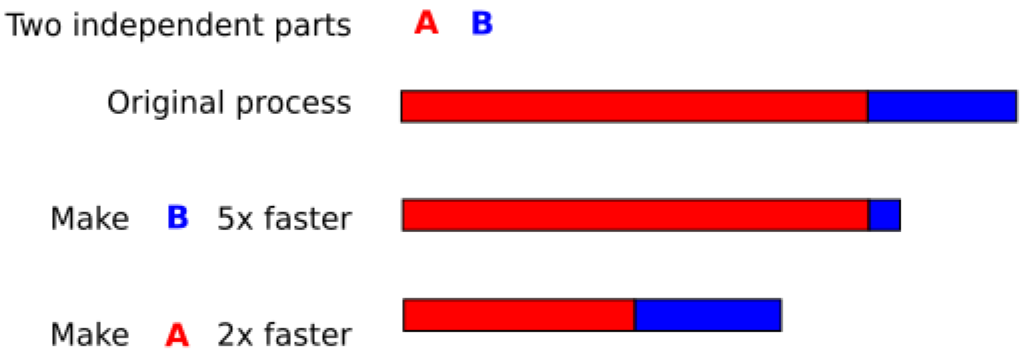
Optimally, the speed-up from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. Any large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. This relationship is given by the equation:

$$S = \frac{1}{1 - P}$$

where *S* is the speed-up of the program (as a factor of its original sequential runtime), and *P* is the fraction that is parallelizable. If the sequential portion of a program is 10% of the runtime, we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."

Gustafson's law is another law in computing, closely related to Amdahl's law. It can be formulated as:



Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. With effort, a programmer may be able to make this part five times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A twice as fast. This will make the computation much faster than by optimizing part B, even though B got a greater speed-up (5× versus 2×).

$$S(P) = P - \alpha(P - 1)$$

where *P* is the number of processors, *S* is the speed-up, and *α* the non-parallelizable part of the process. Amdahl's law assumes a fixed problem size and that the size of the sequential section is independent of the number of processors, whereas Gustafson's law does not make these assumptions.

Dependencies

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Let P_i and P_j be two program fragments. Bernstein's conditions describe when the two are independent and can be executed in parallel. For P_i, let I_i be all of the input variables and O_i the output variables, and likewise for P_j. P_i and P_j are independent if they satisfy

- $I_j \cap O_i = \varnothing,$
- $I_i \cap O_j = \varnothing,$
- $O_i \cap O_j = \varnothing.$

Violation of the first condition introduces a flow dependency, corresponding to the first statement producing a result used by the second statement. The second condition represents an anti-dependency, when the second statement (P_j) would overwrite a variable needed by the first expression (P_i). The third and final condition represents an output dependency: When two statements write to the same location, the final result must come from the logically last executed statement.

Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep(a, b)
2:    c := a·b
3:    d := 2·c
4: end function
```

Operation 3 in Dep(a, b) cannot be executed before (or even in parallel with) operation 2, because operation 3 uses a result from operation 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2:      c := a·b
3:      d := 2·b
4:      e := a+b
5: end function
```

In this example, there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein’s conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.

Race conditions, mutual exclusion, synchronization, and parallel slowdown

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, "threads" is generally accepted as a generic term for subtasks. Threads will often need to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. For example, consider the following program:

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A Write back to variable V	3B: Write back to variable V

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished. Therefore, to guarantee correct program execution, the above program can be rewritten to use locks:

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow a program.

Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock locks multiple variables all at once. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is known as parallel slowdown.

Fine-grained, coarse-grained, and embarrassing parallelism

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

Consistency models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

One of the first consistency models was Leslie Lamport's sequential consistency model. Sequential consistency is the property of a parallel program that its parallel execution produces the same results as a sequential program. Specifically, a program is sequentially consistent if "... the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

Software transactional memory is a common type of consistency model. Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Mathematically, these models can be represented in several ways. Petri nets, which were introduced in Carl Adam Petri's 1962 doctoral thesis, were an early attempt to codify the rules of consistency models. Dataflow theory later built upon these, and Dataflow architectures were created to physically implement the ideas of dataflow theory. Beginning in the late 1970s, process calculi such as Calculus of Communicating Systems and Communicating Sequential Processes were developed to permit algebraic reasoning about systems composed of interacting components. More recent additions to the process calculus family, such as the π-calculus, have added the capability for reasoning about dynamic topologies. Logics such as Lamport's TLA+, and mathematical models such as traces and Actor event diagrams, have also been developed to describe the behavior of concurrent systems.

Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, whether or not those instructions were using a single or multiple sets of data.

Flynn's taxonomy

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

According to David A. Patterson and John L. Hennessy, "Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme."

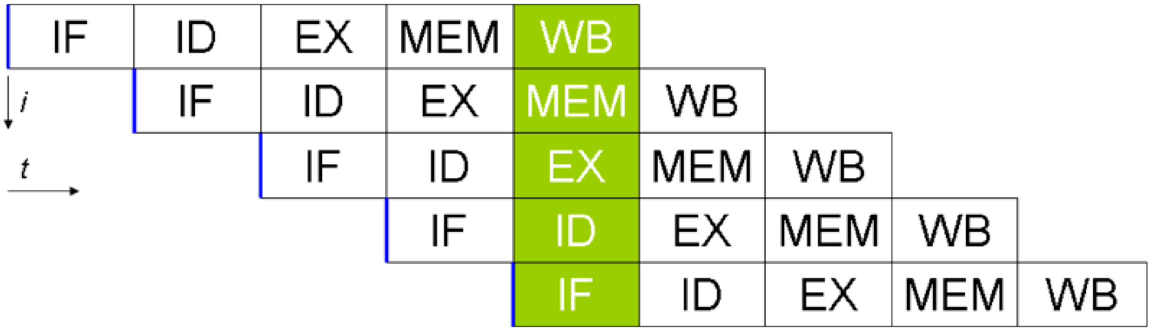
Types of parallelism

Bit-level parallelism

From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.

Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until recently (c. 2003–2004), with the advent of x86-64 architectures, have 64-bit processors become commonplace.

Instruction-level parallelism

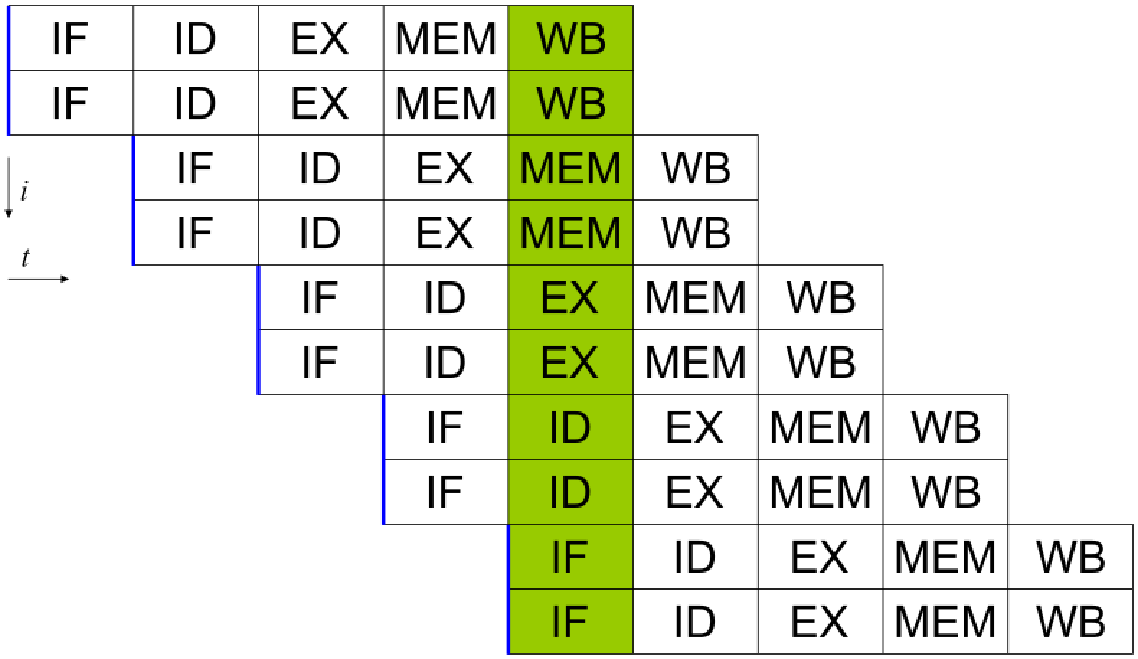


IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

A computer program is, in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch, decode, execute, memory access, and write back. The Pentium 4 processor had a 35-stage pipeline.



A five-stage pipelined superscalar processor, capable of issuing two instructions per cycle. It can have two instructions in each stage of the pipeline, for a total of up to 10 instructions (shown in green) being simultaneously executed.

In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm (which is similar to scoreboarding but makes use of register renaming) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism .

Data parallelism

Data parallelism is parallelism inherent in program loops, which focuses on distributing the data across different computing nodes to be processed in parallel. "Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure." Many scientific and engineering applications exhibit data parallelism.

A loop-carried dependency is the dependence of a loop iteration on the output of one or more previous iterations. Loop-carried dependencies prevent the parallelization of loops. For example, consider the following pseudocode that computes the first few Fibonacci numbers:

```
PREV1 := 0
PREV2 := 1
do:
    CUR := PREV1 + PREV2
```

```
       PREV1 := PREV2
       PREV2 := CUR
    while (CUR < 10)
```

This loop cannot be parallelized because CUR depends on itself (PREV2) and PREV1, which are computed in each loop iteration. Since each iteration depends on the result of the previous one, they cannot be performed in parallel. As the size of a problem gets bigger, the amount of data-parallelism available usually does as well.

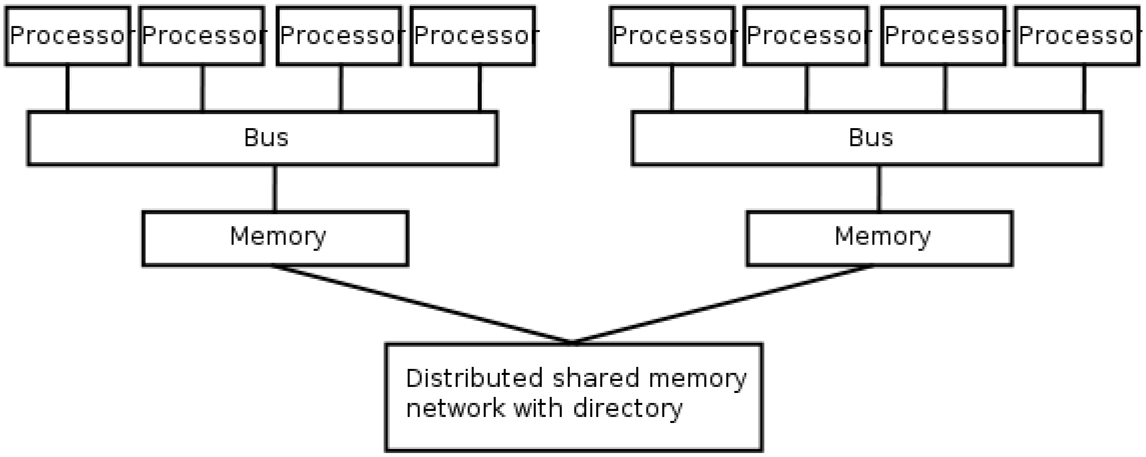
Task parallelism

Task parallelism is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data". This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism does not usually scale with the size of a problem.

Hardware

Memory and communication

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory.



A logical view of a Non-Uniform Memory Access (NUMA) architecture. Processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory.

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a Non-Uniform Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access.

Computer systems make use of caches—small, fast memories located close to the processor which store temporary copies of memory values (nearby in both the physical and logical sense). Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Bus snooping is one of the most common methods for keeping track of which values are being accessed (and thus should be purged). Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared-memory computer architectures do not scale as well as distributed memory systems do.

Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a hypercube with more than one processor at a node), or n-dimensional mesh.

Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.

Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

Multicore computing

A multicore processor is a processor that includes multiple execution units ("cores") on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); by contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. Each core in a multicore processor can potentially be superscalar as well—that is, on every cycle, each core can issue multiple instructions from one instruction stream.

Simultaneous multithreading (of which Intel's HyperThreading is the best known) was an early form of pseudo-multicoreism. A processor capable of simultaneous multithreading has only one execution unit ("core"), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."

Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

Cluster computing

A Beowulf cluster

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. The vast majority of the TOP500 supercomputers are clusters.

Massive parallel processing

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having "far more" than 100 processors. In an MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect."

A cabinet from Blue Gene/L, ranked as the fourth fastest supercomputer in the world according to the 11/2008 TOP500 rankings. Blue Gene/L is a massively parallel processor.

Blue Gene/L, the fifth fastest supercomputer in the world according to the June 2009 TOP500 ranking, is an MPP.

Grid computing

Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, grid computing typically deals only with embarrassingly parallel problems. Many grid computing applications have been created, of which SETI@home and Folding@Home are the best-known examples.

Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common grid computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, grid computing software makes use of "spare cycles", performing computations at times when a computer is idling.

Specialized parallel computers

Within parallel computing, there are specialized parallel devices that remain niche areas of interest. While not domain-specific, they tend to be applicable to only a few classes of parallel problems.

Reconfigurable computing with field-programmable gate arrays

Reconfigurable computing is the use of a field-programmable gate array (FPGA) as a co-processor to a general-purpose computer. An FPGA is, in essence, a computer chip that can rewire itself for a given task.

FPGAs can be programmed with hardware description languages such as VHDL or Verilog. However, programming in these languages can be tedious. Several vendors have created C to HDL languages that attempt to emulate the syntax and/or semantics of the C programming language, with which most programmers are familiar. The best known C to HDL languages are Mitrion-C, Impulse C, DIME-C, and Handel-C. Specific subsets of SystemC based on C++ can also be used for this purpose.

AMD's decision to open its HyperTransport technology to third-party vendors has become the enabling technology for high-performance reconfigurable computing. According to Michael R. D'Amour, Chief Operating Officer of DRC Computer Corporation, "when we first walked into AMD, they called us 'the socket stealers.' Now they call us their partners."

General-purpose computing on graphics processing units (GPGPU)

Nvidia's Tesla GPGPU card

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.

In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, recently several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and CTM respectively. Other GPU programming languages are BrookGPU, PeakStream, and RapidMind. Nvidia has also released specific products for computation in their Tesla series.

Application-specific integrated circuits

Several application-specific integrated circuit (ASIC) approaches have been devised for dealing with parallel applications.

Because an ASIC is (by definition) specific to a given application, it can be fully optimized for that application. As a result, for a given application, an ASIC tends to outperform a general-purpose computer. However, ASICs are created by X-ray lithography. This process requires a mask, which can be extremely expensive. A single mask can cost over a million US dollars. (The smaller the transistors required for the chip, the more expensive the mask will be.) Meanwhile, performance increases in general-purpose computing over time (as described by Moore's Law) tend to wipe out these gains in only one or two chip generations. High initial cost, and the tendency to be overtaken by Moore's-law-driven general-purpose computing, has rendered ASICs unfeasible for most parallel computing applications. However, some have been built. One example is the peta-flop RIKEN MDGRAPE-3 machine which uses custom ASICs for molecular dynamics simulation.

Vector processors

The Cray-1 is the most famous vector processor.

A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. "Vector processors have high-level operations that work on linear arrays of numbers or vectors. An example vector operation is $A = B \times C$, where A, B, and C are each 64-element vectors of 64-bit floating-point numbers." They are closely related to Flynn's SIMD classification.

Cray computers became famous for their vector-processing computers in the 1970s and 1980s. However, vector processors—both as CPUs and as full computer systems—have generally disappeared. Modern processor instruction sets do include some vector processing instructions, such as with AltiVec and Streaming SIMD Extensions (SSE).

Software

Parallel programming languages

Concurrent programming languages, libraries, APIs, and parallel programming models (such as Algorithmic Skeletons) have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture—shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads and OpenMP are two of most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API. One concept used in programming parallel programs is the future concept, where one part of a program promises to deliver a required datum to another part of a program at some future time.

Automatic parallelization

Automatic parallelization of a sequential program by a compiler is the holy grail of parallel computing. Despite decades of work by compiler researchers, automatic parallelization has had only limited success.

Mainstream parallel programming languages remain either explicitly parallel or (at best) partially implicit, in which a programmer gives the compiler directives for parallelization. A few fully implicit parallel programming languages exist—SISAL, Parallel Haskell, and (for FPGAs) Mitrion-C.

Application checkpointing

The larger and more complex a computer is, the more that can go wrong and the shorter the mean time between failures. Application checkpointing is a technique whereby the computer system takes a "snapshot" of the application—a record of all current resource allocations and variable states, akin to a core dump; this information can be used to restore the program if the computer should fail. Application checkpointing means that the program has to restart from only its last checkpoint rather than the beginning. For an application that may run for months, that is critical. Application checkpointing may be used to facilitate process migration.

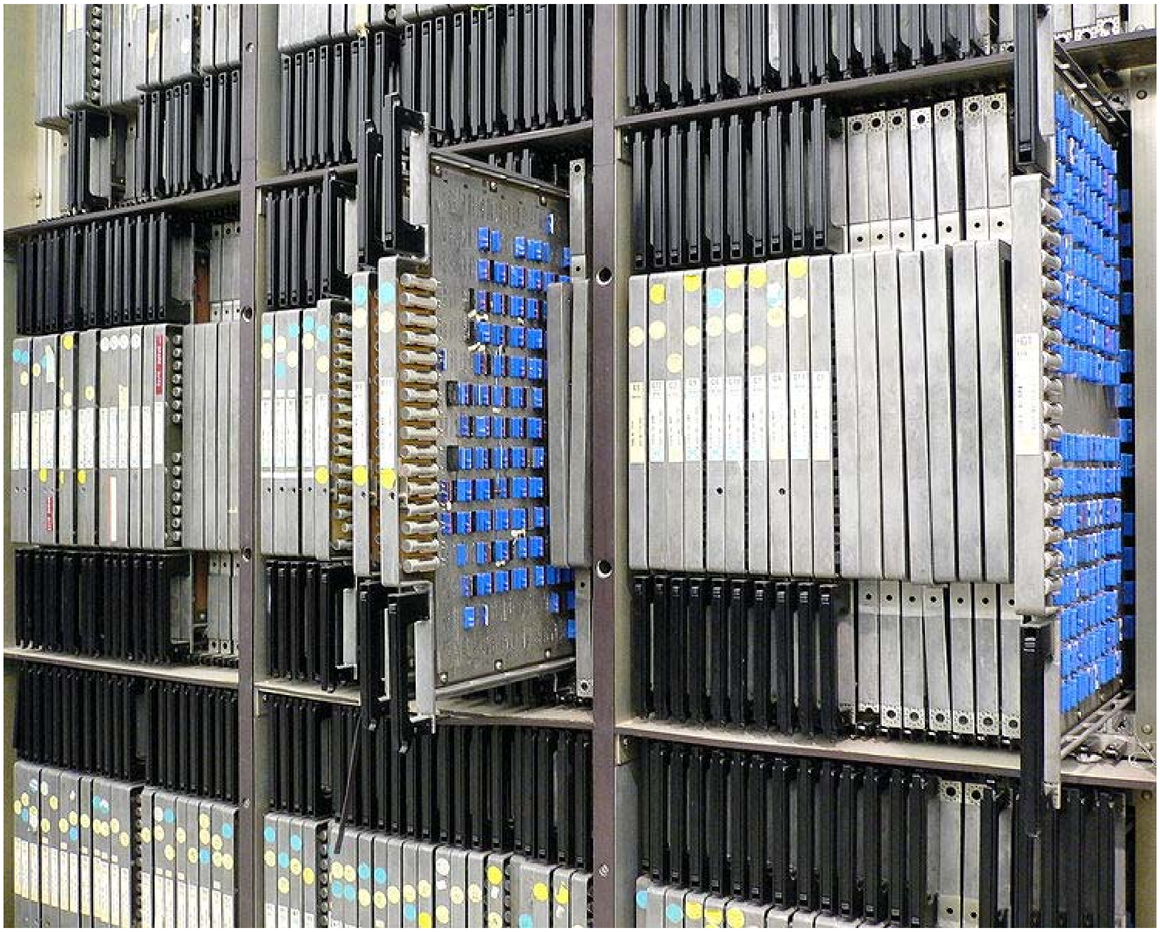
Algorithmic methods

As parallel computers become larger and faster, it becomes feasible to solve problems that previously took too long to run. Parallel computing is used in a wide range of fields, from bioinformatics (protein folding) to economics (mathematical finance). Common types of problems found in parallel computing applications are:

- Dense linear algebra
- Sparse linear algebra
- Spectral methods (such as Cooley–Tukey fast Fourier transform)
- *n*-body problems (such as Barnes–Hut simulation)
- Structured grid problems (such as Lattice Boltzmann methods)
- Unstructured grid problems (such as found in finite element analysis)
- Monte Carlo simulation
- Combinational logic (such as brute-force cryptographic techniques)

- Graph traversal (such as sorting algorithms)
- Dynamic programming
- Branch and bound methods
- Graphical models (such as detecting hidden Markov models and constructing Bayesian networks)
- Finite-state machine simulation

History



ILLIAC IV, "perhaps the most infamous of Supercomputers"

The origins of true (MIMD) parallelism go back to Federico Luigi, Conte Menabrea and his "Sketch of the Analytic Engine Invented by Charles Babbage". IBM introduced the 704 in 1954, through a project in which Gene Amdahl was one of the principal architects. It became the first commercially available computer to use fully automatic floating point arithmetic commands.

In April 1958, S. Gill (Ferranti) discussed parallel programming and the need for branching and waiting. Also in 1958, IBM researchers John Cocke and Daniel Slotnick discussed the use of parallelism in numerical calculations for the first time. Burroughs Corporation introduced the D825 in 1962, a four-processor computer that accessed up to

16 memory modules through a crossbar switch. In 1967, Amdahl and Slotnick published a debate about the feasibility of parallel processing at American Federation of Information Processing Societies Conference. It was during this debate that Amdahl's Law was coined to define the limit of speed-up due to parallelism.

In 1969, US company Honeywell introduced its first Multics system, a symmetric multiprocessor system capable of running up to eight processors in parallel. C.mmp, a 1970s multi-processor project at Carnegie Mellon University, was "among the first multiprocessors with more than a few processors". "The first bus-connected multiprocessor with snooping caches was the Synapse N+1 in 1984."

SIMD parallel computers can be traced back to the 1970s. The motivation behind early SIMD computers was to amortize the gate delay of the processor's control unit over multiple instructions. In 1964, Slotnick had proposed building a massively parallel computer for the Lawrence Livermore National Laboratory. His design was funded by the US Air Force, which was the earliest SIMD parallel-computing effort, ILLIAC IV. The key to its design was a fairly high parallelism, with up to 256 processors, which allowed the machine to work on large datasets in what would later be known as vector processing. However, ILLIAC IV was called "the most infamous of Supercomputers", because the project was only one fourth completed, but took 11 years and cost almost four times the original estimate. When it was finally ready to run its first real application in 1976, it was outperformed by existing commercial supercomputers such as the Cray-1.